Ruby: Unmasking Unsafe Rust in Stripped Binaries via Machine Learning

Xiang Cheng*, Sangdon Park*†, HyungSeok Han‡, Xiaokuan Zhang§ and Taesoo Kim
Georgia Institute of Technology, †Pohang University of Science and Technology, ‡Microsoft, §George Mason University

***Abstract*—Rust, as an emerging system programming language, introduces `unsafe` to allow developers to bypass safety checks during compilation. As a result, 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. Therefore, we propose RUBY, the first tool that unmasks `unsafe` regions in Rust binaries using machine learning. By capturing the subtle differences in the binary instructions, RUBY can identify 91.75% of the total `unsafe` regions with a false positive rate of 6.16%, beating SOTA LLM models including GPT-5.2, Claude-4.5 and Gemini-3. We further applied RUBY to guide symbolic execution and fuzzing, showing a speed-up of 57.95% and 21.26%, with five bugs confirmed and patched by Google in Android library fuzzing.**

***Index Terms*—Rust, Memory Safety, Machine learning**

I. INTRODUCTION

Rust is a rapidly growing systems programming language due to its focus on enhancing memory safety [66] and addressing memory safety vulnerabilities [43, 26]. For example, Rust has been employed in system software such as Mozilla Firefox, Google Chrome [27], Linux kernel modules [58], Windows kernel components [67], and various other device drivers [59].

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 [65] such as dereferencing pointers or interfacing with external C libraries.

Rust Memory Safety Bugs: Source Code Analysis. Despite the memory safety mechanism in Rust, more than 360 memory safety bugs have been found in Rust programs in the last five years [28]. 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 [4, 24]. 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.

*These authors contributed equally to this work.

To find and fix memory-safe bugs in Rust programs, a rich line of prior work (*e.g.,* [6, 80, 44, 39, 13, 40, 32, 11, 83, 17]) has focused on analyzing the unsafe regions in Rust source code. For example, Rudra [6] presented three important patterns of memory safety bugs in unsafe Rust and identified these bugs through static analysis of the unsafe regions; RPG [80] prioritized fuzzing unsafe regions in Rust libraries to more efficiently identify memory safety bugs; ERASAN [44] proposed a more efficient address sanitizer [61] for Rust programs, leveraging the characteristics of unsafe regions.

Our Focus: Rust Binary Analysis. While source-level analysis has advanced significantly in detecting memory safety issues, it is fundamentally constrained by the requirement for source code accessibility. This limitation is increasingly acute as Rust is adopted in commercial and security-critical sectors where proprietary codebases are the norm. For example, when auditing proprietary Rust-based firmware, safety-critical embedded systems, or commercial drivers such as Ferrocene [21] and Windows kernel components [67], security analysts must rely exclusively on the binary.

Furthermore, binary analysis remains essential even when source code is available, as it provides the ultimate ground truth of the executable's behavior. It can unmask vulnerabilities introduced during the compilation pipeline, such as aggressive compiler optimizations that alter memory access patterns [74] or complex macro expansions that generate unforeseen **direct unsafe operations** [10]. These instruction-level risks are often elided or obscured at the source level, making their detection in stripped binaries a vital necessity for robust security assurance.

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 for the following reasons:

- **Diverse unsafe Rust operations:** In practice, the rustc compiler checks 12 unsafe operations [4] outlined in Table I. Each operation presents a distinct root cause, and addressing each unsafe operation constitutes a subproblem of the broader challenge of recovering unsafe Rust from binaries.
- **Ambiguity between safe/unsafe Rust:** 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 1,

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

Listing 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 5.

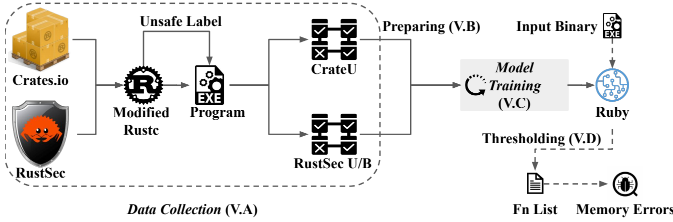

Fig. 1: The workflow of RUBY as an unsafe Rust classifier, details are shown in section V

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. Although the diverse unsafe operations and subtle differences from safe operations make it challenging to unmask unsafe Rust, our **insight** is: there are binary representation differences (see section IV) between safe and unsafe Rust due to compiler optimizations, because safe Rust provides more constraints. These subtle differences can be learned by machine learning models during the large scale training phase [7, 62] and captured through inference. We propose RUBY, which leverages ML to address the challenges mentioned above and identify unsafe regions within Rust binaries.

As the workflow of RUBY shown in Fig. 1, we first construct CrateU dataset, which contains the unsafe labels on functions from the entire crates.io. Next, RUBY collects the RustSecU and RustSecB datasets containing real-world five-year memory safety bugs from Rustsec, 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. With the help of the classification confidence score, RUBY outputs a prioritized list as targets, which can be used by reverse engineers to focus on only 7.43% of programs, with an 88.92% chance of finding potential memory safety bugs. In comparison, a default analysis requires examining 41.38% of the binaries, demonstrating a slowdown of more than 5.5×. We also conducted an evaluation of RUBY against SOTA LLMs including GPT-5.2, Claude-4.5, and Gemini-3; RUBY achieves higher accuracy than these general-purpose models.

To illustrate the real-world application of RUBY, we incorporate RUBY into the program analysis workflow, using its results as guidance for static analysis by Angr [63] and directed fuzzing by AFLGo [9]. 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 patch testing for Rust libraries, we successfully identified five bugs in Android's Rust libraries, which have been confirmed and patched by Google.

In summary, our **contributions** are as follows:

- We propose a new tool RUBY[*], 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 extensively evaluate RUBY, demonstrating that RUBY is **effective** (achieving high accuracy), **efficient** (capable of analyzing crates.io within a week) and **robust** (compatible with x64 and ARM, and different Rustc/LLVM versions).
- We use RUBY to identify five bugs in Android Rust libraries, which have been confirmed and patched by Google.

Ethics Discussion. We collect Rust packages from crates.io and bug information from RustSec reports [28]. All data collected is publicly available, and we only use this data for research purposes. For the five bugs we identified in Android, we reported them to Google, and all bugs have been patched.

II. BACKGROUND

A. Rust Programming Language

Safe Rust. Rust language [42] provides a memory-safety guaranty at compile time while allowing control over low-level access to resources. The claimed memory-safety guaranty is demonstrated in [54, 34] under certain assumptions regarding language models. Specifically, safe Rust is defined as: Safe Rust will never cause undefined behavior [76].

To achieve memory safety: safe Rust leverages *ownership* and *borrowing* to ensure that there are no undefined behaviors in the compiled programs [54]. Ownership is a relationship between a value and a variable. Each value in Rust is owned by *only one* variable, and the 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 the need for a run-time garbage collector. In particular, since a value's associated memory is freed only via `drop()`, double-free or use-after-free bugs are avoided through the compiler's lifetime checks. Although ownership provides a vital safeguard to avoid memory-safety bugs, it is too strict to allow only one owner for each value. Thus, borrowing is introduced to address this issue.

Borrowing allows us to reference 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

[*]Our artifact is available at https://doi.org/10.5281/zenodo.14217066

Label	Unsafe Operations	Description	% of functions
1	CallToUnsafeFunction	Call an unsafe function. This type has two subtypes:	17.61%
2		"internal" (label 1) and "external" (label 2)	0.17%
3	UseOfInlineAssembly	Use a `asm!` 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 const functions. (Deprecated)	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. (Deprecated)	0
11	BorrowOfLayoutConstrainedField	Borrow a layout constrained field. (Deprecated)	0
12	CallToFunctionWith	Call to a function that requires special target features.	1.02%

TABLE I: Twelve unsafe operations named internally in Rustc [68] and their distributions (safe Rust takes the 76.28% [4]). RUBY studies their unique binary representations in section IV.

```
impl<T> Vec<T> {
    ...
    pub unsafe fn set_len(&mut self,
        new_len: usize) {
        self.len = new_len;
    }
    pub const unsafe fn get_unchecked<I>(&self,
        index: I) -> &I::Output
    {
        unsafe { &*index.get_unchecked(self) }
    }
}
fn get(self, slice: &[T]) -> Option<&T> {
    if self < slice.len() {
        unsafe{Some(slice_get_unchecked(slice,
            self))}
    }
    ...
}
```

Listing 2: Example of direct unsafe operations using Rust standard library's `Vec`. The first two functions are *indirect* unsafe operations, and the third function `get()` is the *direct* unsafe operation.

this restriction, safe Rust ensures that no other party can write to the variable when it is borrowed as a mutable reference.

Unsafe Rust. Although safe Rust provides a strong guaranty of memory safety and relatively flexible restrictions, there are many scenarios where 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 Rust compiler's checks inside safe regions and requires programmers to ensure memory safety inside the unsafe regions. In particular, the Rust compiler defines five operations as unsafe operations, which help programmers identify unsafe regions and ensure memory safety [65]: 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 (rustc 1.67) performs explicit checks for 12 distinct unsafe operations [68], as summarized with their relative frequencies in Table I. These 12 explicit operations are derived from the five canonical categories of unsafe operations, and several of them (5, 10, 11) are already deprecated. Among the remaining nine categories of active unsafe operations, we designate those that do not involve (1) and (2) the invocation of unsafe functions as **direct unsafe operations**, because direct unsafe operations are, by themselves, capable of inducing undefined behavior.

Because Rust's safety guarantees are defined at the module level [8], certain functions, such as `set_len`, are declared unsafe yet only cause undefined behavior indirectly, via other direct unsafe operations. An illustrative example is given in Listing 2: the two unsafe functions shown there do not themselves contain any direct unsafe operations in their bodies but instead rely on the direct unsafe operation in the third function, `get`, to potentially violate memory safety. In this work, RUBY primarily targets these direct unsafe operations as exemplified in `get`.

The direct unsafe Rust takes 24.6% of the codebase across all crates in the Rust community [4], 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.

B. Memory Safety Error

Memory safety is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers. Operations that violate this protection are considered memory safety errors [75]. Among these errors, access errors, uninitialized variables, and use-after-free can directly enable malicious users to hijack the program, leading to serious consequences. Therefore, searching for memory safety errors is important for reverse engineers to ensure the program's safety.

On the other hand, these memory safety errors (excluding the memory leaks) are considered undefined behaviors in Rust [76], which are designed to be encompassed by unsafe Rust. Ideally, because of the safe Rust definition, undefined behavior (operations) is considered to be triggered only through direct unsafe operations. Therefore, by helping to reduce the search space for reverse engineers in locating direct unsafe

operations, RUBY can accelerate the memory safety error hunting process.

III. Threat Model

We assume the possession of a Rust binary compiled with the release profile [37], 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 operations in the binary that are written using unsafe Rust and may directly trigger undefined behavior, referred to as **direct unsafe operations**. RUBY outputs such direct unsafe operations as a sub-region of the whole program operations to minimize manual effort in exploitation. Once these direct 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 VI-D).

IV. Binary Representations of Unsafe Rust

We first study each of the unsafe operations in detail and show the unique binary representations*.

A. Calling Unsafe Functions

The first category of unsafe operations in Rust is calling unsafe functions. Based on the owner of the unsafe functions, there are two cases for calling unsafe functions: 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.

1) Calling unsafe Rust Function: Calling unsafe Rust functions (*i.e.,* label 1) indicates 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 problem [35].
- 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 an observation under RUBY's threat model:

Direct Unsafe Delegation: Every unsafe Rust function must eventually ground its unsafety in a direct unsafe operation other than the call 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 I. 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.

Guided by this observation, RUBY intentionally skips the identification of the `call` instruction itself. Instead, it attempts to locate the concrete direct unsafe operations that serve as the root cause of the unsafety. By pinpointing these "root" locations, downstream tools like directed fuzzing can more effectively target the specific instructions where memory corruption is physically possible, rather than wasting resources on high-level function entries.

2) 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 to distinguish the function sources.

B. 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

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

```rust
let mut x: u64 = 4;
asm!(
    "mov {tmp}, {x}",
    "shl {tmp}, 1",
    "shl {x}, 2",
    "add {x}, {tmp}",
    x = inout(reg) x,
    tmp = out(reg) _,
);          // Assembly preserved in binary
let mut y: u64 = 4;
let tmp = y << 1;
y = y << 2;
y = y + tmp; // Compiler optimizes computation
```

Listing 3: 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

```
<reference>:
    ... ; omitted 13 instructions for simplicity
    ; deref reference is optimized and removed
    8eb4: add    rsp,0x40
    8eb8: pop    rbx
    8eb9: ret

<pointer>:
    ... ; omitted 13 instructions for simplicity
    8f33: mov    eax,DWORD PTR [rbx]; deref ptr
    8f35: add    rsp,0x40
    8f39: pop    rbx
    8f3a: ret
```

Listing 5: Different binary instruction outputs for code shown in Listing 1 to access pointers/references. The safe guarantee of reference promotes its value to registers. Check full example at Godbolt.

```rust
#[rustc_layout_scalar_valid_range_start(1)]
#[rustc_layout_scalar_valid_range_end(5)]
struct NonZeroI64(i64);
size_of::<Option<NonZeroI64>> // 8
#[repr(transparent)]
struct PlainI64(i64);
size_of::<Option<PlainI64>> // 16
```

Listing 4: 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

Rust compiler's type and memory safety checks. As shown in Listing 3, 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 for RUBY to identify.

C. 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, niche optimization [5] 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.

D. 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 IV-A, RUBY can leverage characteristic differences between Rust and C/C++ binaries to infer the origins of these static variables and assess their safety.

E. 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 1.

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.

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

Listing 6: 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

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

Listing 7: Example of using AES-NI instructions in unsafe Rust, resulting in `aesenclast` instructions in assembly. Full example can be found at Godbolt

As demonstrated in Listing 5, the compiler optimizes reference-based access into register operations, whereas raw pointer dereferencing produces additional memory access instructions.

F. 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.

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 6, 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.

G. 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 7, the AES-NI instruction `_mm_aesenc_si128` is utilized to enhance the performance of AES encryption. This leads to the inclusion of the `aesenclast` instruction within the binary, which is tailored for the x86_-64 architecture. These hardware-dependent instructions are preserved in the binary and can be identified by RUBY.

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%)

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

H. 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.

V. PROPOSED APPROACH

Building on the insights of binary representations in section IV, we propose RUBY, a machine learning-based tool for detecting unsafe operations. In this section, we present the design choices and implementation details of RUBY.

A. Dataset Collection

Table II 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.,* $\{(x_1, u_1), \ldots, (x_m, u_m)\}$, 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 $u_i \subseteq \{1, ..., 12\}$ for "unsafe" root causes in Table I), 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 [28]. The RustSec dataset is a set of tuples of a function in assembly code, unsafe labels, and a bug label, *i.e.,* $\{(x_1, u_1, y_1), \ldots, (x_n, u_n, y_n)\}$, 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 the RustSecU dataset, and only with bug labels is denoted as the 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.

To evaluate RUBY on real-world vulnerabilities, we collected 360 advisories from the RustSec Advisory Database [28] spanning five years. Two people independently audited these

reports to identify reproducible memory-safety vulnerabilities, filtering the set to 121 unique root-cause bugs by excluding logic-only issues. We downloaded the 257 corresponding crates and compiled them to generate our evaluation dataset, RustSecB. To prevent data leakage, these 257 crates were strictly excluded from the 11M-function CrateU training set.

While the dataset contains 121 unique vulnerabilities, they manifest as 301 buggy function entries in the compiled stripped binaries. This expansion is a technical consequence of the Rust compilation model: a single generic vulnerability is often specialized into multiple concrete instances via *monomorphization* or propagated across function boundaries through *inlining*. In total, the RustSecB evaluation set comprises 1.8M functions, of which only 301 are buggy ($< 0.02\%$). This extreme class imbalance underscores the "needle-in-a-haystack" challenge of binary-level bug hunting and the necessity of RUBY's search-space reduction.

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.

B. Preprocessing

Embedding Metadata. As discussed in section IV, 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.

C. 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 $\mathcal{U} \coloneqq \{2, 3, 4, 6, 7, 8, 9, 12\}$ be a set of unsafe labels, where the corresponding unsafe notations are defined in Table I,

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

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

We consider the following parameterized function of the binary unsafe classifier based on $\hat{s}$:

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

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 [41] as the backbone of the unsafe classifier, which is the long-standing stable masked language model based on transformers [71] 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.

D. 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 [50], where heuristic methods are mostly considered. We consider a rigorous thresholding approach that comes with a PAC guarantee based on conformal prediction [72, 77].

PAC Algorithm. We adopt the PAC conformal set algorithm [47, 48] for thresholding. Let $\bar{\theta}$ be the upper Clopper-Pearson (CP) bound [15], 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 $\mathbb{P}_{k\sim\text{Binomial}(m,\mu)}\left[\mu \leq \bar{\theta}(k;m,\delta)\right] \geq 1-\delta$. 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:

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

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.,* $k := \sum_{(x,u)\in S_{\text{cal}}} \mathbb{1}\left(1-\hat{s}(x,0) < \tau\right)$. 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 $\delta = 10^{-3}$.

VI. 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.

A. RQ1: Unsafe Classifier Evaluation

Setup. 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 evaluation on RustSecU, which are excluded from CrateU to ensure that RUBY never learned from those data.

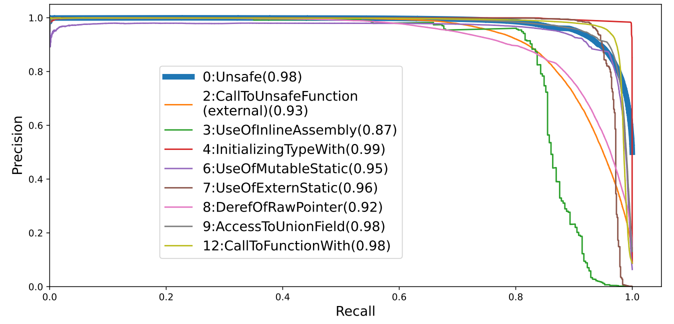

Fig. 2: Precision-recall(AUPRC) evaluation on x64 binaries in CrateU dataset. RUBY achieves relatively high scores on each label and 0.98 on the unsafe detection.

1) Evaluation on CrateU: We first evaluate RUBY over CrateU for the unsafe classification task. Fig. 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 (1) for $\hat{\tau}$, and the 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 guaranty of recall, controlled by ε. In practice, we desire to have a list of functions containing the desired rate of unsafe functions, so we empirically demonstrate that the proposed algorithm achieves this goal.

Comparison with SOTA LLMs. We compare RUBY model with thresholding to the SOTA LLMs, including chat models: GPT-5.2 [45], Claude-Sonnet-4.5 [1], Gemini-3-flash, and reasoning models: Claude-Opus-4.6 [2]. We conduct few-shot (K) and best-of-N evaluations to validate the abilities of the SOTA models with gradually increased context. The few-shot evaluation involves gradually adding K examples for each candidate category in the prompt and evaluating the model's inference based on existing examples. For the best-of-N evaluation, we ask the model N times for the same question and check if there is a chance that the model outputs the correct answer. The evaluation results are shown in Table III.

Overall, RUBY successfully lead the accuracy by around 20% as an dedicated model for unsafe Rust classification. Although few-shot and best of N evaluations help SOTA LLMs to better understand the problem, they still suffer from lack of context and requires further improvement.

Comparison with Fine-tuned Models. To further demonstrate RUBY's benefit in the specific domain of the Rust binary unsafe

Model	K=1			K=3			K=5		
Accuracy % / N=?	1	3	5	1	3	5	1	3	5
Sonnet-4.5	12.3	13.7	13.4	24.0	25.7	22.6	32.0	31.1	30.0
Gemini3-flash	38.6	41.1	41.7	61.4	62.3	62.0	63.7	64.3	**65.1**
GPT-5-chat	43.4	48.3	48.9	55.4	57.7	57.8	58.8	62.6	63.4
Opus-4.6	49.7	50.3	49.4	59.7	60.3	60.9	65.4	65.7	**66.1**
Geimin3-pro	33.1	47.1	52.5	49.7	56.6	62.6	47.4	55.7	60.6
GPT-5.2-pro	44.0	47.7	49.1	54.0	58.3	59.7	57.7	62.3	62.6
GPT-4.1-FT	58.0	62.3	64.0	64.0	69.1	72.3	64.3	69.2	**74.6**
RUBY	83.1								

TABLE III: Accuracy score of RUBY's unsafe classification comparing with SOTA LLM models. We conduct few-shot (K) and best of N evaluations on chat and reasoning models, overall RUBY improves the accuracy score by training on unsafe classification tasks.

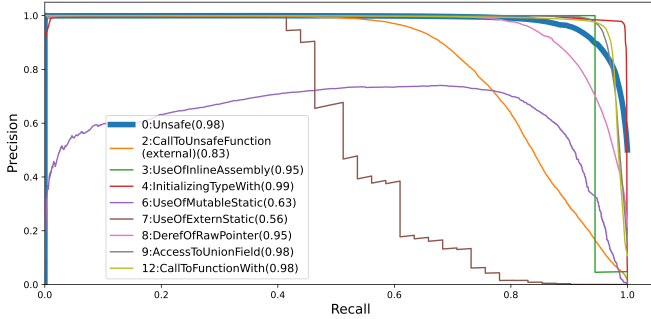

Fig. 3: Precision-recall (AUPRC) evaluation on x64 binaries in RustSecU dataset, RUBY achieves 0.98 AUPRC score with a precision 95.21% and recall 91.09%, similar to the CrateU dataset result.

classification task, we conduct fine-tuning using OpenAI's GPT-4*. For each label, we sampled 20 cases for each unsafe label and combined them into our fine-tuning dataset, reusing the same benchmark that we tested with the vanilla models. The result is shown in the `GPT-4.1-FT` row in the Table III. By comparing the result, we believe it's more efficient for RUBY to adopt a dedicated model instead of general LLM models.

2) 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 Fig. 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.

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

1) 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:

*As of February 2026, gpt-4.1-2025-04-14 is the latest model supporting fine-tuning in OpenAI's services.

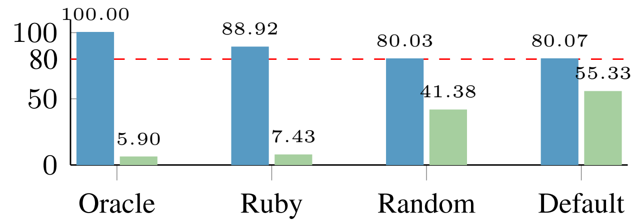

Fig. 4: RUBY's overall performance for narrowing down search space. We set the same target recall value as 80% (blue bar) and compare the number of candidates (green bar, lower value is better) picked by each method.

category	name (LOC)	AUPRC (↑)
Web browser	`servo(11.10M)` [69]	0.890
Ruby interpreter	`artichoke(1.93M)` [3]	0.839
Python interpreter	`rustpython(8.02M)` [20]	0.832
JavaScript runtime	`deno(9.36M)` [20]	0.907

TABLE IV: Evaluation of RUBY's unsafe classification on large applications. RUBY achieves high AUPRC scores on all applications and even 0.907 on `deno`.

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

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 Fig. 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.

2) Performance Overhead: RUBY's performance is affected by the bootstrapping process like decompiling stage and model loading stage. 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.

3) 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 runtime*, 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

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

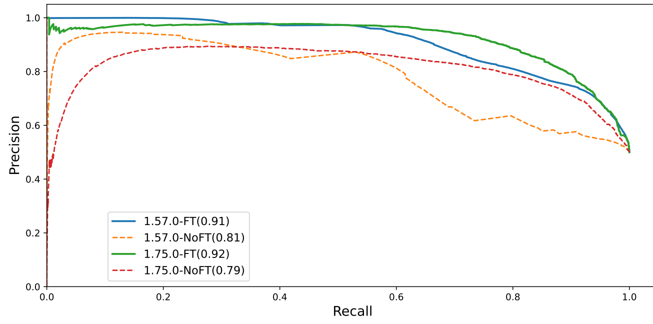

Fig. 5: Precision-recall (AUPRC) evaluation on different compiler versions of unsafe Rust classification. With the help of fine-tuning, RUBY improves the scores to 0.91 for 1.57.0 and 0.92 for 1.75.0.

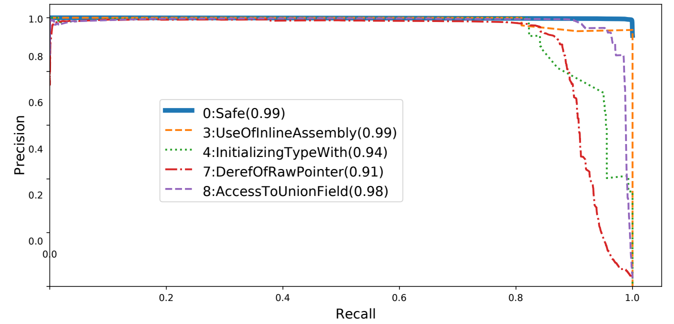

Fig. 6: Precision-recall (AUPRC) evaluation on ARM binaries in CrateU dataset. RUBY achieves high scores as x64, showing its robustness on different architectures.

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

Table IV 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.

C. RQ3: Robustness Evaluation

By the definition of unsafe Rust in subsection II-B, 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).

1) 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 [56], which uses LLVM-15 as the backend. To further demonstrate RUBY's robustness, we collected the RustSecU dataset using Rustc 1.57.0 [55] with the LLVM-13 backend and Rustc 1.75.0 [57] 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 1.75.0, fine-tuned for two epochs in less than an hour.

Result. The comparison of the unsafe classification task is presented in Fig. 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.

2) 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 Fig. 6. RUBY demonstrate high performance on both 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.

3) Ablation Study: **Setup.** RUBY embeds the static analysis result in the input of the model to help the model identify the inner connections. To demonstrate the effectiveness of this approach, we picked the `UseOfMutableStatic` label and retrained it without embedding the static analysis using the same dataset CrateU.

Result. The comparison is shown in Fig. 7, the static analysis helps RUBY improve the AUPRC by 0.61, showing the benefit of embedding the static analysis result into the model input.

D. RQ4: Bug Hunting Applications

We applied RUBY on the Angr, directed fuzzing and Android system fuzzing to assess its end-to-end effectiveness.

1) Angr Analysis: Angr [63] is a powerful binary analysis tool that finds bugs through symbolic execution. However, performing symbolic execution is time and resource-consuming, as it involves the exploration of different paths in the program. Therefore, RUBY can be integrated as a guiding framework, enabling the prioritization of suspect functions.

For instance, in the `predefined` binary of RUSTSEC-2021-0094, there are 1,434 functions in total, and under the default ordering the actual buggy function is ranked 783. RUBY reorders functions according to their estimated unsafe probabilities, elevating the buggy function to rank 140 and accelerating the bug localization process.

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

TABLE V: Angr analysis performance for Rust binaries in different architectures. `TO` 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.

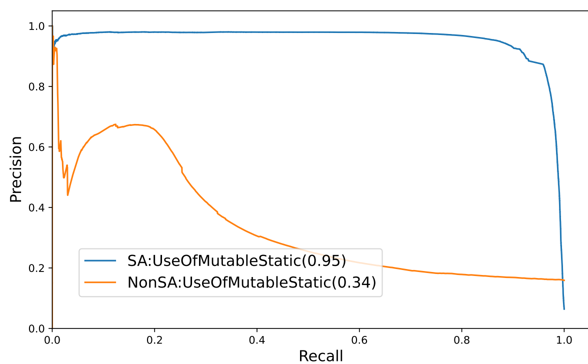

Fig. 7: Ablation study of static analysis on x64 binaries in CrateU dataset. 'SA' means static analysis. The static analysis result helps RUBY improves the performance of target label classification task by 0.61 in AUPRC.

Setup. We collected the benchmark from the RustSec dataset, manually filtered out bugs that were not shown in the binary programs, and successfully built 14 vulnerable binaries. We set up our Angr analyzes based on QueryX [29]'s memory safety analysis script, identifying heap/stack overflows, use-after-free, and out of bounds access 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 V.

Result. 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, RUBY saves 57.95% of time to find the same bugs, largely saving time and resources during the vulnerability hunting process.We also performed the same evaluation on the ARM binaries, and RUBY saves 61.4% on ARM architectures.

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%

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

2) Fuzzing Analysis: Directed fuzzers are emerging dynamic analysis tools that leverage control flow graphs and static analysis to compute the instruction's distance to the target function and prioritize the corpus that reaches closer locations.

Setup. We apply the result of RUBY to a directed fuzzer as its target function to show the direction of RUBY for dynamic analysis. Specifically, we use AFLGO [9] 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 undirected AFL, then apply RUBY to these binaries and launch the AFLGO with RUBY's output as the target. We count the wall time of the fuzzers that encounter the first crash as a result and repeat the fuzzing process three times to obtain the mean value.

Result. The result is shown in Table VI; on average, with the help of RUBY, AFLGO can save 21.26% of the time to find the same bug without guidance.

3) Android Rust Library Fuzzing: To evaluate RUBY's end-to-end effectiveness, we applied RUBY to the Android system and provided guidance for fuzzing Rust libraries.

Setup. As of Android 16.0.0, there are 93 Rust crates serving as fundamental OS libraries, containing 1,534 function APIs.

Testing all of the functions requires a lot of engineering effort. We consider the following steps: given a crate library in `.dylib or .so`, (1) we feed all the functions into RUBY and get a prioritized target list; manually develop the fuzzing harnesses for each function, and (2) conduct black box fuzzing with AFL++ in Frida mode [70] for the top 50 targets. For each target, we run for a maximum of 24 hours.

Result. We identified and confirmed five different 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 and patched by the maintainers. Compared with the default approach of fuzzing each function individually, RUBY saves 74.6% of time to find these bugs.

VII. 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.

Inherited Limitations. 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.

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 [41], which is a classic BERT model specialized for classification tasks. Finally, due to the context window limitation, RUBY may be inaccurate because of different context splits.

VIII. RELATED WORK

Unsafe Rust and its usage. Although software engineers try to avoid `unsafe` regions in their program to avoid potential memory safety problems [25, 18], many Rust crates are using "unsafe" more frequently and lead to implicit usage and wide spread of unsafe blocks in Rust binaries [24]. Also, studies [52, 4] show that these unsafe usages of are often for good or unavoidable reasons, which are not easy to remove. While `unsafe` catches the attention of programmers on memory safety, it can be also used by reverse engineers to find the weakness of the given Rust binaries. To our knowledge, RUBY is the first tool to recover the `unsafe` regions from raw binary instructions.

Binary bug hunting. Approaches to find bugs or vulnerabilities have been proposed through binary analyses [63, 16, 12, 64, 19, 73, 78, 79]. Angr [63] is a powerful framework that combines static and dynamic analyses to automatically find general vulnerabilities in binary executable. In contrast, other tools are designed to find specific bugs in binaries; oo7 [73] is designed for *spectre attacks*, KEPLER [78] is targeted for *control-flow hijacking*, and DTaint [12] aims to detect *taint-style* vulnerabilities. Furthermore, machine learning has been applied to program analyses for bug finding [46, 38, 51]. VulDeePecker is a system that leverages deep learning to automatically detect bugs inside programs. However, [84] also proposed several open questions that may interfere with the performance of applying machine learning to the search for bugs.

General binary analysis. Beyond bug hunting, general binary analysis is an essential task in computer security. Some of this research (*e.g.,* function boundary detection) can be regarded as a basis of RUBY. While Nova [33] utilizes hierarchical attention and contrastive learning for general semantic representation, RUBY is a specialized classifier designed for the security-critical task of identifying **direct unsafe operations**. In *binary-binary code matching*, a binary function or an entire program is represented in a vector to retrieve functions or programs similar to a binary target given [23, 81, 49]. The core of known approaches is learning binary code representation to summarize the instructions of a function or a program into a vector based on recurrent neural networks [31], graph neural networks [60], or transformer-based models [71]. Similarly, *binary-source code matching* finds similar binary or source code given source or binary code [82]. *Function prototype inference* is predicting the type of function (*e.g.,* the number of arguments and the type of argument) given instructions of a function [14]. *Function boundary detection* enumerates the list of the start and end of a function in a binary [30, 62, 36, 22], and *malware classification* classifies each binary regardless of whether it embeds malware [53].

IX. CONCLUSION

In this paper, we present RUBY, the first tool to leverage machine learning and static analysis to identify unsafe regions in Rust binaries. Our evaluation shows that RUBY can identify unsafe regions from Rust binaries with precision 93.84% and recall 91.75%. RUBY can accelerate the vulnerability finding process in symbolic execution (static) and directed fuzzing (dynamic) by 57.95% and 21.26% respectively. By applying RUBY to blackbox fuzzing in Android, we successfully identified 5 unknown bugs in the Rust libraries and all the bugs were confirmed by the developers.

X. ACKNOWLEDGMENT

We thank our shepherd Hui Xu and the anonymous reviewers for their valuable feedback and suggestions. We also thank Mingyu Guan and Zhuoran Yu for their assistance with model training. 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 Facebook, Mozilla, Intel, VMware and Google.

*The bugs are available at: https://issuetracker.google.com/issues/399131919

References

[1] Anthropic. Introducing Claude 4.5 Sonnet.

[2] Anthropic. Introducing Claude Opus 4.6.

[3] Artichoke Ruby. Build the next ruby for wasm with artichoke. https://github.com/artichoke/artichoke.

[4] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[5] Atticus. Niche optimizations in rust. https://www.0xatticus.com/posts/understanding_rust_niche/, July 2024. Accessed: 2025-11-21.

[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] Bryan Beckman and Jed Haile. Binary analysis with architecture and code section detection using supervised machine learning. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 152–156. IEEE, 2020.

[8] Alexis Beingessner and The Rust Project Developers. *The Rustonomicon: Working with Unsafe*. The Rust Project, 2024. Accessed: 2024-05-20.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[10] CasualX. Issue #60: Unsound issue while converting bytes to utf8 str. https://github.com/CasualX/obfstr/issues/60, 2023. Accessed: 2025-10-10.

[11] Hung-Mao Chen, Xu He, Shu Wang, Xiaokuan Zhang, and Kun Sun. Typepulse: Detecting type confusion bugs in rust programs. *arXiv preprint arXiv:2502.03271*, 2025.

[12] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, 2018.

[13] Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang, and Taesoo Kim. RUG: Turbo LLM for Rust Unit Test Generation. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*, Ottawa, Canada, April 2025.

[14] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 99–116, Vancouver, BC, August 2017. USENIX Association.

[15] 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.

[16] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 269–278. IEEE, 2006.

[17] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–21, 2023.

[18] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan 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*, pages 1–13, 2024.

[19] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 53(2):392–404, 2018.

[20] Deno Land Inc. Deno: A modern runtime for javascript and typescript. https://github.com/denoland/deno.

[21] Ferrocene Developers. Ferrocene: Safety-critical systems in rust. https://github.com/ferrocene/ferrocene, 2024. Accessed: 2025-11-29.

[22] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.

[23] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, 2019.

[24] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 246–257. IEEE, 2020.

[25] 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.

[26] 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.

[27] Google Security Blog. Supporting Use of Rust in Chromium. Blog post.

[28] The Rust Secure Code Working Group. The rust security advisory database. https://rustsec.org/.

[29] HyungSeok Han, JeongOh Kyea, Yonghwi Jin, Jinoh Kang, Brian Pak, and Insu Yun. Queryx: Symbolic query on decompiled code for finding bugs in cots binaries. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3279–312795. IEEE, 2023.

[30] hex-rays. IDA pro disassembler. https://hex-rays.com/.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-

term memory. *Neural computation*, 9(8):1735–1780, 1997.

[32] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. Rulf: Rust library fuzzing via api dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 581–592. IEEE, 2021.

[33] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. *arXiv preprint arXiv:2311.13721*, 2023.

[34] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[35] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[36] Hyungjoon Koo, Soyeon Park, and Taesoo Kim. A Look Back on a Function Identification Problem. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2021.

[37] The Rust Programming Language and Contributors. Release profiles. https://doc.rust-lang.org/beta/cargo/reference/profiles.html#release.

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[39] 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.

[40] 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*, pages 234–245, 2020.

[41] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[42] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[43] 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.

[44] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. Erasan: Efficient rust address sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 239–239. IEEE Computer Society, 2024.

[45] OpenAI. Introducing GPT-5.2.

[46] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 450–459. IEEE, 2015.

[47] 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.

[48] Sangdon Park, Edgar Dobriban, Insup Lee, and Osbert Bastani. PAC prediction sets under covariate shift. In *International Conference on Learning Representations*, 2022.

[49] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8476–8486. PMLR, 18–24 Jul 2021.

[50] John Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 1999.

[51] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[52] 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.

[53] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[54] Eric C. Reed. Patina : A formalization of the rust programming language. Technical report, University of Washington, 2015.

[55] Rust Blog. Announcing rust 1.57.0, December 2021. Accessed: 2025-11-02.

[56] Rust Blog. Announcing rust 1.67.0, January 2023. Accessed: 2025-11-02.

[57] Rust Blog. Announcing rust 1.75.0, December 2023. Accessed: 2025-11-02.

[58] Rust for Linux Contributors. Rust-for-Linux/linux.

[59] Rust-GPU Contributors. Rust-GPU/Rust-CUDA.

[60] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[61] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual*

technical conference (USENIX ATC 12), pages 309–318, 2012.

[62] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, Washington, D.C., August 2015. USENIX Association.

[63] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[64] Pengfei Sun, Luis Garcia, Gabriel Salles-Loustau, and Saman Zonouz. Hybrid firmware analysis for known mobile and iot security vulnerabilities. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 373–384. IEEE, 2020.

[65] The Rust Team. Unsafe operations in rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#unsafe-superpowers.

[66] The Rust Team. Rust: A language empowering everyone to build reliable and efficient software. *https://www.rust-lang.org/*, 2010.

[67] The Register. Microsoft to explore Windows 11 code written in Rust. Online article.

[68] The Rust Project Developers. Rust compiler source code: `check_unsafety.rs` (version 1.67), 2023. Commit: c9c57fadc47c8ad986808fc0a47479f6d2043453.

[69] The Servo Project Developers. The servo parallel browser engine project. https://github.com/servo/servo.

[70] Chris Valsamaras. Fuzzing Android binaries using AFL++ Frida Mode. Blog post on Medium, May 2024. Accessed: 3 December 2025.

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[72] Vladimir Vovk. Conditional validity of inductive conformal predictors. *Machine learning*, 92(2-3):349–376, 2013.

[73] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

[74] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 1–7, 2012.

[75] Wikipedia contributors. Memory safety. https://en.wikipedia.org/wiki/Memory_safety, Accessed: 2025.

[76] Wikipedia contributors. Undefined behavior. Wikipedia, The Free Encyclopedia, Accessed: 2025.

[77] Samuel S Wilks. Determination of sample sizes for setting tolerance limits. *The Annals of Mathematical Statistics*, 12(1):91–96, 1941.

[78] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, 2019.

[79] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

[80] 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.

[81] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.

[82] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 3872–3883. Curran Associates, Inc., 2020.

[83] Yehong Zhang, Jun Wu, and Hui Xu. Rumono: Fuzz driver synthesis for rust generic apis. *ACM Transactions on Software Engineering and Methodology*, 34(6):1–28, 2025.

[84] Yang Zhao, Xingzhong Du, Paddy Krishnan, and Cristina Cifuentes. Buffer overflow detection for c programs is hard to learn. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 8–9, 2018.