

Dynamic Analysis of Unsafe Rust: Behaviors and Benchmarks

Jiaye Fang

The George Washington University
Washington, DC, USA

Arsalan Bin Najeeb

The George Washington University
Washington, DC, USA

Dylan O'Neill

The George Washington University
Washington, DC, USA

Prince Noah Johnson*

University of Notre Dame
Notre Dame, Indiana, USA

Zhixiao Zhang

The George Washington University
Washington, DC, USA

Jie Zhou

The George Washington University
Washington, DC, USA

Abstract

Rust permits the use of *unsafe* code, which can violate both memory safety and concurrency safety. Prior work investigates unsafe code in Rust projects and provides insights into what, how, and why it is used. However, these studies provide only *static* metrics about unsafe code, while lacking an understanding of its *dynamic* behaviors at execution time. In parallel, many security mechanisms have been proposed to mitigate unsafe code in Rust. But the evaluations of these tools are suboptimal due to the lack of a benchmark suite that emphasizes unsafe code.

In this paper, we present the first empirical study of the runtime behavior of unsafe code in real-world Rust projects. We develop a dynamic analysis framework that programmers can use to answer key questions about unsafe code at execution time. Moreover, using the dynamic analysis framework, we curate the first benchmark suite specifically designed for evaluating the performance and memory overhead of security defenses against unsafe code in Rust.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; *Software safety*; • **General and reference** → *Empirical studies*.

Keywords: Unsafe Rust, Memory Safety, Dynamic Analysis, Benchmark Suite

ACM Reference Format:

Jiaye Fang, Arsalan Bin Najeeb, Dylan O'Neill, Prince Noah Johnson, Zhixiao Zhang, and Jie Zhou. 2026. Dynamic Analysis of Unsafe Rust: Behaviors and Benchmarks. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026)*, November 15-18, 2026, Hong Kong, China. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3828523.3857071>

*Work done at the George Washington University.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11

<https://doi.org/10.1145/3828523.3857071>

1 Introduction

Rust [27] has emerged as one of the most promising systems programming languages because it provides both strong safety guarantees and high performance. However, due to the low-level nature of systems programming and Rust's limitations—both in language design (e.g., its strict ownership model [27]) and in compiler implementation (e.g., the inability to reason about the safety of certain complex programs [43])—Rust can be overly restrictive. To address this, Rust allows *unsafe* code that enables low-level operations beyond the compiler's verification capabilities, providing significant flexibility for systems programming. This mechanism, unfortunately, introduces security risks, especially memory safety vulnerabilities [62].

Prior studies investigate Rust projects to understand the use of unsafe code [7, 16, 44, 50, 66], including how frequently unsafe code appears in Rust source code [7, 16] and what unsafe features are widely used [7, 44]. They find that unsafe code is commonly used and provide insights into how and why unsafe Rust code is used, helping Rust programmers and language developers better understand and constrain unsafe behavior.

Yet, these studies have only focused on *static* measurements, while lacking a *dynamic* understanding of unsafe code—i.e., *how unsafe Rust code behaves at execution time*. This distinction is important for two reasons. First, unsafe code may not execute in proportion to its static representation. For example, prior work [7, 16] counts calls to unsafe functions, a major unsafe code pattern. An unsafe call site on a hot path, however, contributes far more runtime exposure than one on a cold path. Second, some measurements, such as memory usage, are only possible at runtime. Therefore, dynamic measurements can reveal actual unsafe code execution, such as how often it runs, what memory it accesses, and what operations dominate execution. These runtime measurements can help programmers prioritize security audits, selectively enforce security policies, and help tool builders gauge the precision of their static analyses.

In addition, the lack of understanding of unsafe Rust at runtime undermines the development of security mechanisms for Rust. Many approaches have been proposed to mitigate

unsafe code in Rust, such as memory isolation [9, 28, 33] and memory sanitizers [13, 42]. These mechanisms incur performance and memory overhead, yet their evaluations suffer from a common limitation: lack of proper benchmarks. Existing work selects benchmark programs merely based on their popularity. This is understandable because Rust is still relatively new and no standard benchmark suites exist. But popularity does not imply suitability. Many popular Rust programs (e.g. base64 [40]) adopted by unsafe Rust defenses contain little or no unsafe code, making them less effective for evaluating security mechanisms. A Rust benchmark suite that emphasizes unsafe code at runtime is needed.

In this paper, we address both of the above problems using dynamic analysis. We present the *first* empirical study of the dynamic behavior of unsafe code in real-world Rust programs. Our study centers around two key questions: (1) *What is the runtime footprint of unsafe code, and how does it differ from its static occurrence?* and (2) *What security-related insights can dynamic analysis of unsafe code provide?* This study complements prior static measurements and provides a broader picture of unsafe Rust code in practice.

We develop a compiler-based dynamic analysis framework to assess unsafe Rust code along these dimensions: execution time, heap memory usage, executed instructions, functions containing unsafe code, and the impact of unsafe code introduced by the standard libraries into the user crates. We examined about 1,200 crates ranked by download count and selected 100. The dataset covers all common Rust program types. We also develop an LLM-based workflow to synthesize additional workloads for crates with insufficient ones. Our study shows that unsafe code generally has a larger runtime footprint than its static appearance suggests, while this footprint is highly non-uniform across crates and can even dominate execution in some cases. We also find that unsafe code introduced from the standard libraries substantially increases unsafe runtime exposure.

Furthermore, we curate the *first* Rust benchmark suite, named URBench, that emphasizes unsafe code. Using the dynamic analysis framework, we select 17 crates from the examined dataset that exhibit heavy use of unsafe code, along with one additional crate with prominent unsafe usage. Additionally, we tune the workloads of built-in benchmarking functions and develop additional ones for some benchmarks to ensure sufficient execution time and coverage of unsafe code. These 18 benchmarks collectively (1) encompass a wide range of Rust program types, (2) demonstrate intensive unsafe code usage at execution time, and (3) include all common unsafe code patterns. As a result, this benchmark suite is well suited for evaluating Rust safety mechanisms.

In summary, this paper makes three contributions:¹

- We conduct the first empirical study of the execution-time behavior of unsafe code in real-world Rust projects,

Listing 1. Example of Unsafe Rust Code

```

1 extern "C" { fn strlen(s: *const i8) -> usize; }
2
3 struct SafeBuffer<T> { ptr: *mut T, len: usize }
4 impl<T> SafeBuffer<T> {
5     fn get(&self, index: usize) -> Option<&T> {
6         if index < self.len {
7             unsafe { Some(&*self.ptr.add(index)) }
8         } else { None } } }
9
10 unsafe fn process_data(ptr: *mut i32) { ... }
11
12 fn foo() {
13     let mut data = [1, 2, 3, 4, 5];
14     let mut num = 42;
15     let raw_ptr = &mut num as *mut i32;
16     let c_str = CString::new("Hi!").unwrap();
17
18     unsafe {
19         *data.as_mut_ptr().add(1) *= 2; // Ptr
20             arithmetic and raw ptr dereference
21         process_data(raw_ptr); // Unsafe fn call
22         strlen(c_str.as_ptr()); // FFI call
23     }
24
25     let buffer = SafeBuffer { ... };
26     buffer.get(1); // Safe API wrapping unsafe
27 }
```

highlighting both the differences from its static presence and high-exposure unsafe patterns. This study complements prior investigations on unsafe code in source code and compiler IR.

- We present a dynamic analysis framework that enables Rust developers and tool builders to assess and understand the runtime behavior of unsafe Rust code.
- We curate URBench, the first Rust benchmark suite emphasizing unsafe code and suited for evaluating performance and memory overhead of unsafe Rust defenses.

2 Background on Unsafe Rust

The Rust compiler guarantees both memory safety and thread safety [53]. In low-level systems programming, however, there are situations where the safety rules enforced by the compiler can be too restrictive for programmers or overly conservative—meaning that an actually safe program may be rejected because the compiler is unable to infer its safety [27]. Additionally, runtime security checks inserted by compilers can impose expensive performance overhead in certain scenarios, especially in real-time systems.

To address these issues, Rust allows *unsafe code* to circumvent the compiler’s default safety rules [27]. A function or an arbitrarily large portion of a function, once labeled with the `unsafe` keyword, can perform dangerous operations that are not checked by the compiler. Indeed, Rust’s type system is a superset of C’s, and thus unsafe code can freely perform arbitrary unsafe operations as in C, such as raw pointer dereferences and unsafe type casts [4, 27].

Listing 1 illustrates several common unsafe Rust operations: pointer arithmetic (line 19), raw pointer dereference

¹Artifact: <https://github.com/GWSysSec/unsafe-rust-dyn>

(line 19), calling an unsafe function (line 20), an FFI call to a C library function (line 21), and a safe abstraction that wraps unsafe operations behind a bounds-checked interface (lines 4–8). These operations can lead to vulnerabilities. Raw pointer arithmetic, for example, may produce an out-of-bounds pointer, causing a buffer overflow if the pointer is later dereferenced.

These capabilities are often preferred—or even inevitable—for tasks such as implementing certain data structures (e.g., doubly linked lists) or interfacing with foreign code (especially C/C++) [7, 16, 44]. In unsafe blocks, however, safety becomes the programmer’s responsibility rather than relying on the compiler. The Rust security principles advocate using unsafe code *sparingly* and with careful *encapsulation* [4], that is, minimizing and wrapping unsafe code under safe abstractions to ensure that unsafe operations are contained and cannot be misused by client code [7]. For example, in Listing 1, the `SafeBuffer` struct encapsulates a raw pointer with bounds checking in its `get()` method (lines 5–9). But this practice is not always followed, increasing the likelihood of introducing security vulnerabilities [7, 15, 44, 62, 65].

3 Methodology

Prior studies [7, 16, 44, 45] assess unsafe Rust at both the source code and the Mid-level Intermediate Representation (MIR) [37, 43] levels. They investigate the frequency and types of unsafe code, as well as the reasons for using it. We complement prior studies by using dynamic analysis to (1) characterize the runtime behavior of unsafe code and how it differs from its static representation, and (2) investigate the security implications of the analysis of unsafe code.

We focus the measurements on the main crate of each program in our dataset. When analyzing a single program, assessing the main crate and all its dependencies can provide a holistic view of the program’s unsafe behavior. For an empirical study that aggregates results across many programs, however, including dependencies may introduce biases because certain library crates (e.g., `hashbrown`) are common dependencies and execute a large amount of unsafe code, so the same library code may be repeatedly counted and receive disproportionate weight in the aggregate results. We therefore use the measurements of the main crates as our primary results and provide the summarized results with dependent crates included as supplementary data for reference.

Next, we define six research questions that guide our analysis (§3.1). We also describe the criteria for selecting the programs in our dataset (§3.2) and the method for synthesizing additional workloads for them (§3.3).

3.1 Research Questions

Rust’s type system is designed to guarantee both memory safety and concurrency safety, as long as unsafe code is not used [24, 38]. As Section 2 describes, however, Rust’s

type system is a superset of C’s, which opens the door to memory safety and thread safety vulnerabilities [8, 41, 65]. To understand unsafe code at execution time, we measure its resource consumption, identify and classify dynamically executed unsafe instructions and functions, and count the unsafe code introduced by Rust’s standard libraries.

3.1.1 Resource Consumption. We assess the exposure window of unsafe code from the perspective of resource consumption using two metrics: execution time (temporal) and heap memory usage (spatial).

RQ1: How long does unsafe code run? Prior work finds that programmers generally strive to use unsafe code sparingly and cautiously [7], suggesting that such code is often critical for program functionality and performance. Unsafe code, however, may not execute in proportion to its presence in the source code. We therefore measure the execution time of both unsafe code and the entire program to quantify the runtime exposure of unsafe code.

RQ2: How much heap memory does unsafe code access? Unsafe code can not only *unrestrictedly* violate Rust’s memory safety guarantees itself, but also corrupt program state, e.g., safety metadata of data types, thereby introducing undefined behavior in safe Rust code [9, 25]. Therefore, understanding the memory affected by unsafe code is important for security mechanisms, such as in-process memory isolation [6, 9, 33]. Over-approximation of such memory, however, is a common problem: many safe objects, along with accesses to them, are misclassified as unsafe. This is because security defenses often rely on conservative and imprecise static analysis to identify unsafe memory objects [9, 25, 42].

We measure the number and size of unsafe heap objects, as well as the memory accesses to them at runtime. We focus on the heap because most memory safety bugs in Rust programs are heap-related [17, 44, 65]. This analysis can inform the development of security mechanisms that rely on static analysis, as it can highlight the gap between the unsafe heap objects and accesses identified at compile time and those confirmed during execution.

3.1.2 Executed Code. We measure dynamically executed unsafe instructions and the functions that contain them. This analysis characterizes the execution patterns of unsafe code at a low level.

RQ3: How frequently are unsafe instructions executed? Rust is an advanced language that provides high-level abstractions to express complex semantics in concise syntax. As a result, the number of low-level unsafe instructions, both those in the compiler IR or executable and those dynamically executed, may not be proportional to their representation in the source code. Similar to our measurement of the execution time (RQ1), we count the executed unsafe and total instructions. This measurement can help develop security mechanisms, e.g., by helping estimate the runtime cost of enforcing certain checks on unsafe instructions.

RQ4: What are the executed unsafe instructions? To complement **RQ3**, we measure the types of executed instructions. In particular, we focus on memory instructions (load and store), pointer arithmetic, function calls, and type casts, as these correspond to pointer dereferences, manipulation of pointer values, calls to unsafe functions, and unsafe type casts, which are all common sources of vulnerabilities in Rust [7, 44, 65]. Together with **RQ3**, these two analyses provide insight into how unsafe instructions are used in Rust programs at runtime, enabling programmers to verify assumptions about the behavior of unsafe code at a low level and guide further security improvement.

RQ5: How frequently are functions with unsafe code executed? In addition to **RQ3** and **RQ4**, we measure the frequency of executed functions that contain unsafe code. This analysis includes two queries: (1) the proportion of distinct functions executed that contain unsafe code, and (2) the proportion of total function invocations that contain unsafe code. Together, these two metrics capture both the breadth and intensity of unsafe function usage during execution. This analysis helps narrow the scope of functions that may require closer examination.

3.1.3 Impact of Rust’s Standard Libraries. Early during compilation, the Rust compiler often copies the bodies of common APIs, such as `slice::get_unchecked()`, from the standard libraries into the target program, enabling LLVM to inline or optimize them. These included library functions may contain unsafe code, whose impact we measure.

RQ6: How impactful is unsafe code incorporated from Rust’s standard libraries? In general, Rust’s standard libraries (e.g., `std` and `alloc`) are well-audited and less buggy than other crates. The RustBelt project [39] has formally verified the safety of the implementation of certain key data types (e.g., `Rc`) despite their use of unsafe code. Yet, unverified unsafe code remains, and security vulnerabilities have continued to surface in these libraries [59–61, 65]. Such code can be introduced into compiled crates when the compiler copies commonly used small functions from the standard libraries. While our study focuses on user crates, we also assess the impact of unsafe code incorporated from the standard libraries. To this end, we conduct our analysis of **RQ1–RQ5** both with and without accounting for unsafe code originating from the standard libraries. Note that we do not directly analyze the libraries themselves.

3.2 Dataset Selection

Empirical studies need sufficient subjects for statistical validity. However, our investigation requires *executing* programs, which is usually more time-consuming than static analysis,² and may require manual configurations to build and run some crates. Therefore, we must limit the size of our dataset

²While advanced static analyses, such as pointer analysis, can be extremely time-consuming, large-scale code studies usually do not perform them.

Table 1. Program Selection Criteria for Dynamic Analysis

Metrics	Criteria
Popularity	Top downloads on crates.io
Size	At least 1K SLOC
Timeliness	Updated since March 2023; dataset cutoff May 2025
Usability	Compilable and functional source code
Unsafe Code	Containing unsafe code which is invoked by workloads

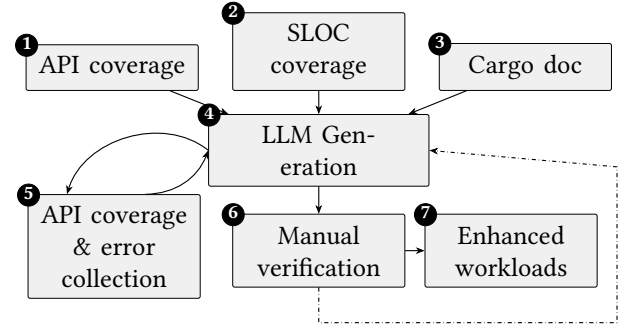


Figure 1. Workflow of LLM-based workload enhancement.

while ensuring its representativeness. We examined around 1,200 popular crates and selected 100 based on the criteria listed in Table 1. Our dataset includes all common types of Rust programs, such as data structures and system utilities. We discuss the selection biases in Section 7 and provide the full list of analyzed crates in the artifact.

3.3 Workload for Dynamic Analysis

The validity of dynamic analysis heavily depends on the workloads of the programs under study. Since our dataset comprises only the most widely used and well-maintained crates, many of them have built-in realistic workloads. For example, `arrayvec` includes a set of integration tests that cover 82% of its public APIs, invoking operations (e.g., `push` and `pop`) that fall into both normal code paths and error handling such as out-of-bounds accesses. Some crates, however, lack sufficient workloads.

To address this limitation, we use Large Language Models (LLMs), assisted by dynamic analysis and manual inspection, to enhance the workloads of programs in our dataset. Figure 1 illustrates the workflow. Because all crates in our dataset are libraries, we use API coverage—which identifies public APIs exercised by a crate’s workload—as a key metric to guide workload enhancement. We treat uncovered APIs as workload gaps and use LLMs to generate additional driver code and inputs to increase coverage.

Steps 1–3 in Figure 1 outline the initial process of collecting API data and SLOC. Steps 1 and 2 use `llvm-cov` [5] to identify API and source code usage at runtime. The workflow then, in Step 3, uses `cargo doc` [56], Rust’s documentation tool, to identify all APIs, as well as their descriptions for later

use. In addition, during Step 3, the workflow supplements this API information by scanning other documentation provided by the developers, such as `README.md` and `TESTS.md`.

The workflow then compares the findings from Steps 1–2 with those from Step 3 to identify APIs uncovered by built-in workloads. These APIs, along with their documentation (e.g., method signature and trait information), are fed into the LLM for workload generation, beginning Step 4. Step 5 checks the generated workloads by compiling and executing them in a continuous feedback loop with Step 4. The loop captures any compilation or execution errors and re-runs generation, while also gathering new API and SLOC coverage data. During Step 5, the LLM also continues to use the information collected in Steps 1–3 to correct errors (e.g., calling an obsolete API) caused by LLM hallucinations. The workflow retries this process up to three times for each set of APIs. We choose three tries to strike a balance between improving API coverage and limiting computational cost.

We perform Step 6 manually by examining the source code of the workloads to check whether they represent realistic usage (workload descriptions available in the artifact). We also check the generated code that failed Step 5 after three attempts and manually rerun generation for those APIs. The LLM is provided with the failed code snippets, their error messages, additional information from Steps 1–3, and our additional guidance in the prompt. Overall, this workflow improves API coverage from 76.4% to 90.4% for our dataset.

4 Analysis Framework Implementation

We build our analysis framework on Rust-1.80.0. Our analysis operates on LLVM IR [29]—a mid-end compilation target for Rust—because it is relatively low-level and thus well suited for investigating the research questions in Section 3.1. We develop LLVM passes and a supporting runtime library. We also extend the Rust compiler to preserve safety-related metadata so that our analysis can identify unsafe instructions. In this section, we describe the measurement passes (§4.1) and discuss a challenge in identifying unsafe code and how we address it (§4.2).

4.1 LLVM IR Passes and Runtime Library

We schedule the passes at the end of the LLVM IR optimization pass pipeline to minimize their interference with the compiler’s standard optimizations. The passes instrument programs to call helper functions defined in the supporting library that records dynamic data during execution.

RQ1: CPU Cycle Counter. We count the CPU cycles elapsed during both the execution of unsafe code and the entire crate. We use CPU cycles rather than regular timers because unsafe code sequences are usually short [7]. The LLVM pass inserts a pair of x86 RDTSCP instructions [22] around each sequence of unsafe instructions to record the CPU’s timestamps. To improve precision, the pass adds a pair of

LFENCE instructions [22] around each RDTSCP pair to isolate the measured instructions by serializing the CPU pipeline. Similarly, we use a pair of RDTSCP and LFENCE around each call to external library functions to exclude time spent in dependencies from our measurement of the main crate.

RQ2: Heap Usage Tracker. We measure the total amount of heap memory allocated and the amount accessed by unsafe code. To track dynamic heap allocations and deallocations, we implement wrapper functions for the system’s default memory allocator using Rust’s `GlobalAlloc` trait [3]. We also implement a compiler pass that instruments unsafe load and store instructions to record which heap objects they access. Finally, we use Rust’s `dtor` macro [36] to report heap usage during the program’s termination phase.

RQ3–RQ5: Instruction and Function Counters. Our compiler tracks all executed instructions, the subset that is unsafe (RQ3), their types (RQ4), and the functions they belong to (RQ5). Note that LLVM IR does not preserve all source-level casts, particularly after the introduction of opaque pointers [35]. We count four LLVM cast instructions: `BitCast`, `PtrToInt`, `IntToPtr`, and `AddrSpaceCast`.

RQ6: Impact of Included Standard Library Code. We add a compiler flag to control whether to mark unsafe instructions originating from the standard libraries as unsafe.

Our analysis framework can be easily extended with new passes. It marks all unsafe instructions, allowing analysis passes to identify them, and existing passes such as the memory tracker can be reused to assist new analyses.

4.2 Identifying Unsafe Instructions

Challenge. Our measurement passes (§4.1) must identify instructions originating from unsafe source code. Safety metadata for unsafe code is available in the Rust compiler’s High-level IR (HIR) [43], but this metadata is discarded during code generation from HIR to Mid-level IR (MIR) [43] and is therefore unavailable later in LLVM IR. Following prior work [13, 25, 33], we address this problem by attaching custom LLVM metadata [34] to instructions generated from unsafe code.³

However, compiler optimizations [2] can still threaten the preservation of unsafe metadata. For example, `instcombine` may replace instructions with new ones [2], potentially losing their safety metadata. Ideally, any transformation pass should propagate unsafe metadata from unsafe instructions to newly generated instructions derived from them. But implementing this is impractical, as it would require understanding and updating *all* transformation passes—more than 100 in total—across 220K lines of code.⁴

³Prior work [13, 25, 33, 42] was built on older compilers in which unsafe metadata was available in MIR. Our work uses a more recent compiler that no longer preserves unsafe metadata in MIR; we therefore also preserve unsafe metadata from HIR to MIR.

⁴This estimate is based on the SLOC count of the `llvm/lib/Transforms` directory in LLVM-18.

Unsafe Instruction Marker. We observe that most optimizations transform instructions *in place*, meaning that they usually remove instructions or replace them with new instructions at the same program point, while passes that reorder instructions are much less common. This implies that when unsafe instructions are transformed, the newly generated, semantically equivalent instructions will most likely remain at the original program point.

Based on this observation, we develop an Unsafe Instruction Marker pass to help our analysis passes identify unsafe instructions. This pass runs at the beginning of the IR pass pipeline, when all original unsafe instructions are present. It inserts a pair of `nop` inline assembly instructions [1, 34] around each sequence of unsafe instructions, marking their boundaries. It sets the `hasSideEffects` flag [34] of these markers to prevent subsequent passes from optimizing them away. Our measurement passes can then use these markers to identify unsafe instructions—either original or newly generated—between each marker pair.

Prior work [13, 25, 33, 42] has also faced the challenge of losing track of unsafe instructions. Yet, to the best of our knowledge, we are the first to explicitly address it.

5 Empirical Results

This section presents the empirical results for the research questions introduced in Section 3.1. We first describe the experimental setup (§5.1), and then we present the empirical data (§5.2). Finally, we discuss its security implications (§5.3). We discuss the threats to the validity of the experimental results in Section 7.

5.1 Experimental Setup

All experiments were conducted on an Ubuntu 22.04 operating system running on a machine equipped with an Intel Xeon w5-3433 processor (16 cores) and 32 GB of DRAM. For the LLM-based workload generation, we used GPT-5.5.

We follow the criteria summarized in Table 1 to select Rust crates for our study. We use each crate’s built-in tests—including unit tests and integration tests—and our LLM-generated drivers for API invocation. For each crate, we calculate a combined result by aggregating the outcomes of its individual runs (e.g., summing the elapsed CPU cycles).

Our customized compiler, which is based on Rust-1.80.0 and augmented with the dynamic analysis passes (§4), compiles all crates and their tests under two configurations: without and with counting unsafe code copied from the standard libraries, to address RQ6 (§3.1.3).

5.2 Results for Research Questions

We present the results both excluding and including unsafe code incorporated from the standard libraries. We focus on

Table 2. Summary of unsafe source code and its execution.

	Min	Geomean	Median	Max
Unsafe SLOC	0.01%	1.5%	1.9%	55.7%
CPU Cycles in Unsafe Code				
Config	Min	Geomean	Median	Max
w/o std libs	0.01%	5.9%	12.2%	98.4%
w/ std libs	0.05%	23.5%	41.8%	99.6%

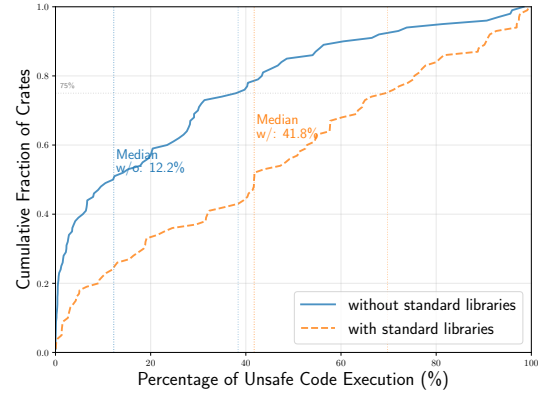


Figure 2. Cumulative distribution of unsafe code execution.

the former in Sections 5.2.1–5.2.4 and discuss the latter in Section 5.2.5. Additionally, Section 5.2.6 summarizes the results when dependent crates are also instrumented.

5.2.1 Temporal Impact: CPU Cycles (RQ1). To quantify the temporal impact of unsafe code execution, we measure the CPU cycles spent on unsafe code (§3.1.1). Table 2 summarizes both the percentages of unsafe source code and its CPU cycle consumption. It shows that, overall, unsafe code accounts for a small fraction of the total execution time (5.9% without considering impacts from standard libraries), yet it executes much more frequently than its appearance in source code (1.5%). This result is expected because unsafe code is often opted for in performance-critical components [7, 16], which tend to execute more frequently than others.

This pattern is further supported by the outlier cases that execute a disproportionately large amount of unsafe code relative to its static occurrence. For example, `memchr` executes 73.8% unsafe cycles whereas its unsafe source code proportion is 14.1%. Manual inspection of `memchr`’s source code and LLVM IR reveals that this high utilization mainly stems from its extensive use of SIMD operations, raw pointer arithmetic for calculating byte offsets, and unaligned memory loads to efficiently handle arbitrary buffer alignments. In contrast, some crates (e.g., `polling` and `value-bag`) show essentially no user-crate unsafe code on hot paths (below 0.02%).

Table 3. Summary of heap memory (by size) accessed by unsafe code and the percentages of unsafe memory instructions (load and store) targeting the heap.

	Without standard libraries					With standard libraries				
	Memory by Size				Mem	Memory by Size				Mem
	Small	Medium	Large	Total	Inst	Small	Medium	Large	Total	Inst
Min	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Geomean	0.8%	1.9%	11.7%	0.7%	0.8%	4.6%	5.0%	32.4%	4.9%	0.4%
Median	0.1%	0.0%	0%	0.6%	1.0%	6.6%	1.1%	13.4%	5.2%	0.7%
Max	81.1%	71.1%	100.0%	81.0%	27.8%	99.8%	76.9%	100.0%	99.8%	27.5%

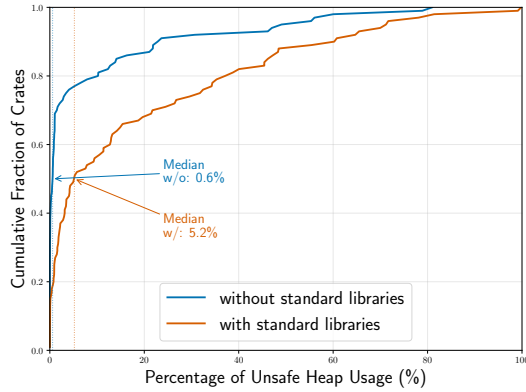


Figure 3. Cumulative distribution of unsafe heap usage.

In addition, Figure 2 presents the cumulative distribution of unsafe code execution percentages. It exhibits a highly skewed distribution with a steep initial gradient and a long tail. 48% of crates spend less than 10% of their execution time in unsafe code, with a median of 12.2% and outliers extending to 98.4%.

Takeaway: Unsafe code accounts for a small portion of execution time (5.9% on average), yet it is moderately higher than its occurrence in the source code (1.5%). Certain crates extensively run unsafe code (over 80%) for performance.

5.2.2 Spatial Impact: Heap Usage (RQ2). We measure unsafe heap memory—i.e., memory that is accessed by unsafe code—to estimate its spatial impact (§3.1.1). Following the object-size classes of glibc’s ptmalloc allocator [23], we group heap objects into three size ranges: small ($\leq 1K$), medium ($1K-128K$), and large ($\geq 128K$). Table 3 reports the percentages of unsafe heap objects across these size classes. It also reports the share of memory instructions (loads and stores) that access unsafe objects. Note that such memory instructions may include those in the safe code. Additionally, Figure 3 depicts the cumulative distribution of unsafe heap usage.

Overall, only a small fraction of heap memory is touched by unsafe code: a geometric mean of 0.7% once standard libraries are excluded. The pattern is size-dependent, rising with object size from a geometric mean of 0.8% for small objects to 1.9% for medium and 11.7% for large ones, indicating that unsafe code is more likely to access large objects.

As with execution time (Figure 2), the distribution is skewed toward the low end: excluding standard libraries, 81% of crates fall below 15% unsafe heap usage and 21% show none at all, leaving a long tail dominated by a few outliers. Compared with Figure 2, Figure 3 shows that unsafe code has a more concentrated spatial footprint in terms of heap usage. Many crates with active unsafe code execution perform non-heap unsafe operations such as hardware feature detection.

The high-usage crates we inspected (e.g., `encoding_rs`, `simdutf8`, `memchr`, `tokio`) share a clear profile: they are performance-critical, low-level libraries built as zero-cost abstractions, using unsafe code for manual memory layout, SIMD intrinsics, unaligned access, and pointer-based iteration to reach C-level performance behind a safe public API.

Use in Rust Defenses. This heap-usage analysis is particularly useful for assessing the precision of Rust defenses that require identifying unsafe heap objects and accesses. TRUST [9], for example, uses pointer analysis [52] to identify unsafe memory objects and isolate them from safe ones. Because pointer analysis is conservative [18, 20], however, TRUST overapproximates the true unsafe set of memory objects and accesses. For example, TRUST reports that, for bytes, 43% of memory reads are unsafe [9], whereas our measurement shows only 6.4%. Similarly, XRust [33], another defense that relies on static analysis, reports that 79% of allocations are unsafe for bytes, while our measurement reports 5.9%. Our heap usage analysis can therefore help tools such as TRUST and XRust quantitatively assess how closely their unsafe identification aligns with dynamic ground truth.

Takeaway: Unsafe heap usage is limited and exhibits size-dependent patterns: larger objects are usually accessed by unsafe code more frequently than smaller ones. Moreover, certain crates extensively use unsafe APIs for fine-grained control over memory management and direct accesses.

Table 4. Frequencies of unsafe instructions. Percentages for instruction types are relative to all unsafe instructions. The last row shows the percentages of unsafe instructions relative to all instructions.

Inst Type	Static counts w/o std libs				Dynamic counts w/o std libs				Dynamic counts with std libs			
	Min	Geomean	Median	Max	Min	Geomean	Median	Max	Min	Geomean	Median	Max
Load	0.0%	9.7%	9.3%	75.0%	0.0%	9.0%	10.4%	46.7%	0.6%	9.5%	12.2%	45.2%
Store	0.0%	6.0%	5.2%	33.7%	0.0%	23.8%	28.1%	57.4%	0.0%	30.3%	30.0%	91.7%
Ptr arith	0.0%	12.0%	12.0%	34.6%	0.0%	5.7%	7.6%	37.5%	0.1%	5.8%	7.8%	52.9%
Cast	0.0%	1.4%	1.1%	16.6%	0.0%	0.0%	0.0%	6.1%	0.0%	0.1%	0.3%	4.0%
Call	0.0%	15.8%	13.9%	100.0%	0.0%	15.4%	15.5%	88.7%	0.0%	10.0%	13.2%	29.4%
Others	0.0%	41.3%	48.3%	100.0%	3.4%	32.6%	36.8%	97.6%	4.3%	34.6%	36.5%	78.7%
Total	0.0%	3.4%	4.9%	61.5%	0.0%	1.8%	1.6%	32.3%	0.7%	18.3%	19.6%	49.0%

5.2.3 Unsafe Instructions (RQ3 & RQ4). We measure executed unsafe instructions by frequency and type and compare them with their static counts (§3.1.2). Table 4 summarizes the results. On average, 1.8% of executed instructions are unsafe, lower than the corresponding static percentage of 3.4%, and 80% of crates execute less than 10% unsafe instructions. The static percentage, however, exceeds the proportion of unsafe code in the source (1.5%), suggesting that unsafe code generally involves more complex code patterns that lower into more instructions than safe code.

As described in Section 3.1.2, we focus on instruction types that are closely associated with common vulnerabilities, such as buffer overflows and type confusions, and emphasize them in Table 4. The static counts are consistent with prior studies [7, 16, 66]: unsafe calls and raw pointer manipulations are the most common reasons for using unsafe code. Notably, stores execute 2.6 times more frequently than loads in unsafe code, inverting both their static ratio and the ratio observed in compute-intensive C/C++ programs (around 2–4× more loads than stores) [10, 51]. Executed unsafe memory instructions also account for a much larger portion of all unsafe instructions than their static representation. Since memory instructions are much more expensive than other types, this helps explain why only 1.8% of executed instructions account for 5.9% of CPU cycles (Table 2).

Takeaway: Executed unsafe instructions account for only 1.8% of all instructions, much lower than their CPU usage (5.9%). Unsafe store instructions are unusually frequent, accounting for 23.8% of all executed unsafe instructions.

5.2.4 Unsafe Code in Functions (RQ5). Complementing the instruction-level view (§5.2.3), we examine unsafe code at the function level. Excluding standard-library code, 3.5% of functions contain unsafe code, but only 2.9% of functions execute unsafe code, falling to 0.4% when weighted by call count. Together, these results suggest that although functions with unsafe code are common, they are often sparse along hot paths. This analysis can also help reveal safe abstractions

Table 5. Execution frequency of functions with unsafe code. **Distinctive** counts each function only once; **Cumulative** counts function invocations cumulatively.

	Static	Distinctive		Cumulative	
	w/o std	w/o std	w/ std	w/o std	w/ std
Min	0.0%	0.0%	8.1%	0.0%	1.2%
Geomean	3.5%	2.9%	24.1%	0.4%	41.2%
Median	1.4%	4.4%	25.8%	2.0%	41.5%
Max	86.0%	26.0%	61.1%	30.0%	100.0%

that wrap unsafe code. For example, several container crates, including hashlink, hashbrown, and smallvec, extensively execute functions containing unsafe code (21%–26%, cumulatively), while exposing only safe APIs.

Takeaway: Functions containing unsafe code are common, but they are often not on hot paths. Certain crates extensively rely on safe abstractions that hide unsafe code.

5.2.5 Impact of Standard Libraries (RQ6). Sections 5.2.1–5.2.4 show that many incorporated standard library functions heavily use unsafe code. This aligns with one of Rust’s core principles: unsafe code should be encapsulated behind safe APIs and verified by library authors [7]. When conservatively treated as untrusted, however, this unsafe code significantly affects program safety across multiple dimensions, including execution time (5.9% vs. 23.5%), executed instructions (1.8% vs. 18.3%), and heap usage (0.7% vs. 4.9%).

To see what drives the inflation, we inspected the source and LLVM IR of the most-affected crates and identified four common unsafe patterns: atomic CPU-feature detection, memory allocation primitives, SIMD intrinsics, and low-level memory movement. These APIs are essential to many crates’ performance, and often to their core functionality, rather than merely supportive.

Takeaway: Unsafe code from Rust’s standard libraries is incorporated into and extensively executed in user crates. It is crucial for both performance and functionality.

5.2.6 Impact of Dependent Crates. As Section 3 states, our analyses focus on the main crate of each program but also measure each crate with its dependencies instrumented.

Overall, including unsafe code from dependent crates increases the aggregate unsafe execution modestly. Excluding unsafe code incorporated from the standard libraries, the geometric means of unsafe execution rise from 5.9% to 7.2% of CPU cycles, from 1.8% to 2.5% of executed instructions, and from 0.7% to 0.8% for unsafe heap usage. When unsafe code from standard libraries is included, the shares of CPU cycles (23.5% vs. 23.1%) and executed instructions (18.3% vs. 19.2%) barely move, while unsafe memory usage increases from 4.9% to 8.5%, demonstrating again that the unsafe code incorporated from standard libraries has a large impact.

These findings are not surprising. The overall increase in unsafe execution can be mainly attributed to the fact that several crates are not only common dependencies but also execute unsafe code frequently, thereby contributing heavily to the aggregate unsafe execution. For example, `memchr` is a dependency of 37 crates in our dataset, and 15.3% of its executed instructions are unsafe, compared with 3.4% on average across the dataset.

When analyzing a specific crate, however, including its dependencies can help because the dependencies may account for most of the program’s unsafe execution. In `serde_yaml`, for example, 5.2% of the main crate’s executed instructions are unsafe, whereas this percentage increases to 63.9% when all dependencies are measured. This substantial difference mainly comes from a single dependency, `unsafe-libyaml`, a translation of the C `libyaml` to Rust, with 224 of its 256 functions declared unsafe. Therefore, analyzing all dependencies together can provide more useful information than assessing the main crate alone.

5.3 Security Implications of Dynamic Analysis

Our dynamic analyses measure runtime exposure of unsafe code rather than the vulnerability surface. A frequently executed unsafe block is not necessarily more likely to contain a vulnerability than a rarely executed one, but if it does, the bug is more likely to be triggered in practice. Our study and analysis framework quantify the runtime exposure of unsafe code: what unsafe code executes and how frequently, what memory it accesses, and whether it is hidden behind safe APIs. These analyses complement static metrics, helping programmers prioritize security audits and estimate the potential cost of defenses against unsafe code.

As our results show, crates with a small fraction of unsafe source code may have a much larger runtime footprint. Conversely, many functions containing unsafe code are reached but are not frequently invoked. This distinction, along with

the executed unsafe code types, can guide security audits and policy enforcement. For example, we find that unsafe SIMD intrinsics are frequently executed in many crates, but they are less likely to lead to vulnerabilities than general code patterns such as pointer arithmetic [15, 48]. Additionally, prior work finds that cold paths are often more likely to contain bugs due to limited testing [11, 58]. Thus, uniformly applying security hardening to all unsafe code may incur unnecessary overhead, whereas a more targeted enforcement guided by dynamic results can keep overhead within budget.

Our dynamic analysis can also quantify unsafe execution behind safe APIs, both in user crates and, partially, in the standard libraries. As discussed in Section 5.2.4, some crates extensively execute unsafe code while exposing only safe APIs; our analysis identifies where such unsafe-backed abstractions are exercised at runtime. Prior work further finds that misuse of APIs in the standard libraries is a major source of memory safety vulnerabilities in Rust [15]. Together, these findings suggest that auditing only unsafe APIs may miss vulnerabilities hidden behind safe APIs, and dynamic analysis can help highlight such cases.

In addition, Section 5.2.2 shows that dynamic analysis can inform the development of security defenses by exposing program behavior at runtime. In particular, our heap analysis identifies unsafe heap objects and their accesses at runtime, providing tool builders with measurements for assessing the precision of static unsafe-object and unsafe-access identification [9, 33, 42].

Finally, our dynamic analysis can help unsafe Rust defenses evaluate their benchmarks. We use our analysis framework to curate a benchmark suite that emphasizes unsafe code execution and present it in Section 6.

6 Benchmarks for Unsafe Rust Defenses

Because unsafe code is commonly used in Rust programs, as shown by prior work [7, 16, 44] and confirmed by this study, and because of its severe security implications [48, 65], many approaches have been proposed to mitigate the risks of unsafe code, such as memory isolation [9, 26, 33, 47] and bug detection [13, 42]. These tools incur performance and memory overhead on the protected programs and use *popular* Rust crates as benchmarks for evaluation. Many such benchmarks, however, contain little or even no unsafe code. For example, `base64` [40]—one of the most widely used crates—was used for evaluation by recent defenses [9, 13, 42]. But `base64` contains no unsafe code. Moreover, even crates that contain unsafe code may not invoke it under their default benchmark setup. To date, a benchmark suite that emphasizes unsafe Rust code is lacking, making the evaluation of unsafe Rust mitigation mechanisms less effective.

In this section, we present URBench, a Rust benchmark suite that emphasizes unsafe code. We select 17 benchmarks from the 100 crates based on the dynamic analysis results

Table 6. Characteristics of benchmarks. **Type** lists program types; D: Data structure, Pa: parser, F: FFI bindings, S: String processing, C: codec, A: Async programming, P: Parallelism, O: Others. **Pattern** lists unsafe code patterns; P: Performance optimization, F: FFI compliance, R: Raw memory manipulation, T: Unsafe type cast, C: Concurrency implementation, B: Bypassing the borrow checker. Percentages of instruction types are relative to all unsafe instructions, while **Total** is relative to all executed instructions. **Fn** cumulatively counts executed functions with unsafe code.

Benchmark	Type	Unsafe			CPU Cycles	Heap Usage	Instructions				Fn	Time
		SLOC	Pattern	Coverage			Load	Store	Call	Total		
arrayvec	D	220 (17.0%)	R,B,T	72.3%	41.1%	0.0%	0.0%	12.3%	2.5%	11.1%	100.0%	31s
async-task	A	425 (32.8%)	C	33.0%	44.4%	99.9%	18.1%	5.3%	26.9%	36.3%	86.1%	28s
getrandom	O	387 (24.2%)	P,F,R,T	58.3%	79.5%	0.0%	27.9%	0.0%	72.1%	3.7%	100.0%	59s
hashbrown	D	805 (7.7%)	B	72.8%	42.2%	94.8%	12.7%	8.0%	31.2%	26.4%	99.3%	250s
httparse	Pa	627 (20.9%)	P,F,R	41.3%	0.8%	0.0%	12.8%	22.0%	31.4%	3.7%	6.3%	141s
jni	F	392 (7.5%)	P,F,R,B	39.1%	1.5%	20.4%	21.4%	5.7%	33.4%	5.4%	35.3%	114s
jpeg-decoder	C	529 (12.6%)	P	30.2%	8.1%	10.6%	14.4%	10.0%	43.6%	4.8%	0.9%	893s
matrixmultiply	O	1.8K (55.7%)	P,F,R,T,C	48.3%	42.8%	99.0%	15.7%	6.8%	54.7%	72.0%	31.6%	203s
memchr	S	1.2K (14.1%)	P,T	25.3%	10.2%	17.6%	14.9%	17.2%	25.2%	9.1%	0.0%	547s
ndarray	D	951 (5.6%)	P,R,T,B	52.3%	41.9%	3.6%	13.7%	3.2%	50.2%	16.4%	96.7%	1025s
parking_lot	P	165 (4.6%)	P,R	31.7%	2.4%	27.1%	7.0%	25.2%	13.3%	1.0%	0.0%	62s
petgraph	D	44 (0.3%)	R	64.2%	32.3%	24.3%	14.9%	9.1%	31.5%	3.2%	49.1%	411s
rayon	P	346 (8.4%)	P,R,T	7.8%	36.4%	4.3%	12.5%	8.3%	57.1%	4.3%	85.0%	225s
ring	O	577 (3.6%)	P,F,R,C	0.0%	31.4%	51.9%	3.2%	0.0%	68.7%	0.2%	0.1%	168s
semver	Pa	145 (10.4%)	R	45.3%	62.3%	99.6%	11.4%	1.4%	38.2%	1.7%	46.6%	14s
simd-json	Pa	1.3K (6.1%)	P,F,R,T,C,B	37.6%	22.2%	44.3%	16.0%	18.8%	34.4%	19.8%	2.8%	939s
smallvec	D	398 (15.3%)	R,T	66.9%	20.6%	5.8%	0.4%	3.9%	19.3%	1.2%	21.4%	164s
tokio	A	1.8K (3.2%)	P,R,T,C	51.3%	1.1%	87.4%	23.2%	9.7%	42.6%	4.4%	27.2%	1297s
Min		44 (0.3%)		0.0%	0.8%	0.0%	0.0%	0.0%	2.5%	0.2%	0.0%	14s
Max		1.8K (55.7%)		72.8%	79.5%	99.9%	27.9%	25.2%	72.1%	72.0%	100.0%	1297s
Geomean		8.9%		41.3%	15.6%	13.9%	11.0%	5.5%	31.3%	5.6%	6.2%	

(§5). In addition, we curate one more real-world Rust project, bringing the total to 18 benchmarks. Note that this benchmark suite is intended for evaluating the performance and memory overhead of Rust security defenses. Evaluating their security effectiveness, however, is important but orthogonal to the purpose of this suite. We describe the benchmark selection (§6.1) and their dynamic characteristics (§6.2). We also compare our benchmark suite with those used in recent Rust defenses (§6.3) and present a case study (§6.4).

6.1 Benchmark Selection

We follow the principles recommended by v. Kistowski et al. [57] and Huppler [21] to curate the benchmark suite. Specifically, it adheres to the following standards:

- **Relevance:** The benchmark suite is designed to evaluate defense mechanisms against unsafe code in Rust. Therefore, it must include a sufficient amount of unsafe code, and this code should be exercised during execution.
- **Fairness:** Since Rust is a general-purpose language, the benchmark suite should cover a broad range of program types, particularly those susceptible to security risks. Moreover, it should encompass all common risky code patterns.

- **Usability:** The benchmarks should be usable across common platforms, i.e., OS- and architecture-independent. In addition, they should be easy to build and run.
- **Reproducibility:** The suite should be self-contained to enable reproduction and easy to verify execution status.

Table 6 lists all the selected benchmarks. To make the benchmarks *relevant* to unsafe code, we select Rust crates with a high percentage of both unsafe source code and dynamically executed instructions. Most well-maintained crates include benchmarking functions, which act as entry points to invoke code of the target crate [54]. However, a key challenge is that many crates do not exercise substantial unsafe code, either because their benchmarking functions do not call (directly or indirectly) functions containing unsafe code, or because their default configurations do not invoke enough unsafe code. Therefore, to improve execution time and unsafe code coverage, we tune the configurations and write additional benchmarking functions for some benchmarks. Benchmark configurations are available in the artifact.

To address the *fairness* concern, our benchmark suite comprises a variety of widely used program types. The second

column of Table 6 lists these categories, including data structures, parsers, FFI bindings, string processing, codecs, asynchronous and parallel programming, and cryptography. All of these crates have significant security implications. For example, codecs and parsers are prone to memory safety errors due to their heavy use of memory buffers, and the security of common data structures impacts the entire program due to their pervasive use.

We also manually examine the *executed* unsafe code in the selected benchmarks to ensure that the suite covers common risky code patterns [7, 16] (summarized in the **Pattern** column in Table 6). Important patterns include, but are not limited to, interfacing with FFI calls, raw pointer and memory manipulation, and disabling safety checks for performance.

Regarding *usability*, our benchmark suite includes only crates that are independent of specific operating systems and architectures, and we use Rust’s Cargo.lock for each benchmark—which contains the exact information about the target crate’s dependencies [55]—to ensure its characteristics are *reproducible*. Finally, we rely on Cargo to automatically *verify* execution status and report errors.

6.2 Dynamic Characteristics of Benchmarks

We use research questions 1–5 (§3.1) and the dynamic analysis framework (§4.2) to analyze the execution-time characteristics of URBench. Recall that the analysis framework measures unsafe Rust code in its CPU consumption, unsafe heap usage, executed instructions, and functions containing these instructions. We use each benchmark’s built-in benchmarking functions, augmented with additional ones we wrote for some benchmarks, as the workloads. Table 6 presents the results.

The percentages of unsafe source code range from 0.3% to 55.7%, with a geometric mean of 8.9%. On average, 41.3% of unsafe code is executed during benchmarking. Similar to the 100-crate empirical study (§5.2), unsafe code in these benchmarks generally has a larger runtime footprint than its static presence suggests, but with two notable differences. First, the relative increase in unsafe execution time is less pronounced for the benchmarks (15.6% vs. 8.9%) than for the 100 crates (5.9% vs. 1.5%); the selected benchmarks already contain a high static density of unsafe code. Second, unsafe heap usage is significantly higher in the benchmarks than in the 100 crates (13.9% vs. 0.7%), as the suite includes multiple crates with exceptionally high unsafe heap usage.

URBench exhibits high diversity not only in program types and unsafe code patterns (see the second and fourth columns of Table 6) but also in the dynamic characteristics of unsafe code. CPU usage of unsafe code, for example, ranges from 0.8% to 79.5%. Certain crates, such as `matrixmultiply`, show significant unsafe usage across CPU time (48.3%), heap (42.8%), and executed instructions (72.0%). In contrast, programs such as `memchr` spend a high fraction of CPU cycles

Table 7. Comparison of our benchmarks with those used in recent Rust defenses. **Eval(uated)** excludes standard-library benchmarks and benchmarks that do not compile. **Cov(erage)** reports the percentage of unsafe source code lines covered during benchmarking.

	Count	Eval	Unsafe				
			SLOC	Cov	CPU	Heap	Inst
ERASan [42]	23	16	2.5%	30.8%	0.9%	1.3%	1.1%
MetaSafe [25]	19	6	3.2%	33.8%	1.8%	6.1%	1.3%
RustSan [13]	20	10	0.9%	22.6%	0.02%	0.08%	0.02%
TRust [9]	11	7	1.3%	35.8%	0.08%	0.6%	0.08%
Our Suite	18	18	8.9%	41.3%	15.6%	13.9%	5.6%

in unsafe code (25.3%) but exhibit relatively modest unsafe heap usage (10.2%) and executed instructions (9.1%).

Importantly, we observe that benchmark relevance cannot be captured by a single metric of unsafe code. For example, unsafe SLOC alone is a weak proxy for dynamic behavior: `petgraph` has only 0.3% unsafe SLOC but its unsafe code accounts for 64.2% of CPU cycles and accesses 32.3% of heap memory, while `memchr` has 14.1% unsafe SLOC but executes only 9.1% unsafe instructions. CPU time, heap usage, and instruction counts expose different aspects of unsafe execution, and our benchmark suite stresses these aspects to different degrees. This multidimensional diversity helps offset biases introduced by specific program types and is essential for a general benchmark suite intended to evaluate different types of unsafe Rust defenses.

6.3 Comparison with Existing Benchmarks

As mentioned above, prior Rust defenses choose benchmarks for performance and memory overhead evaluation based on their popularity, rather than following the common standards described in Section 6.1. But these benchmarks often contain little unsafe code (e.g., `base64`) or lack diversity in program types. We use our analysis framework to assess the benchmarks adopted by four recent Rust defenses [9, 13, 25, 42] and compare them with URBench.

Table 7 summarizes the results. URBench performs substantially more unsafe execution than those used by recent Rust defenses. Our suite spends 15.6% of CPU cycles in unsafe code, compared with at most 1.8% in prior suites, and executes 5.6% unsafe instructions, compared with at most 1.3%. Its unsafe heap usage is also much higher than all compared suites, with a smaller margin over `MetaSafe`, whose result is based on only six benchmarks and is thus less representative.

In addition, our suite covers a wider range of program types than prior suites (Table 6). For example, `MetaSafe`, `RustSan` and `TRust` all lack parallel benchmarks, and all the compared works lack FFI-binding programs. Therefore, URBench is a more effective evaluation tool for assessing unsafe Rust mitigation techniques.

6.4 Case Study

We conducted a case study using our benchmark suite to evaluate the performance overhead of ERASan [42], a Rust memory sanitizer. ERASan builds on ASan [49] and instruments memory accesses to detect memory safety bugs, thereby incurring performance overhead on instrumented programs. ERASan used an older Rust compiler (1.64) than ours (1.80), which cannot compile many of our benchmarks.⁵ In addition, its benchmarking interface is highly customized and requires significant effort to adapt to new benchmarks. After adapting the compatible benchmarks, we ran five to completion and one partially.

We evaluate ERASan’s runtime overhead by comparing the execution time of benchmarks enhanced by ERASan with the execution time of the baseline benchmarks. We ran the benchmarks three times and computed the average across those runs. Overall, we observe significant overhead on all benchmarks: `semver` at 112.6%, `ndarray` at 101.3%, `matrixmultiply` at 117.4%, `arrayvec` at 112.1%, `smallvec` at 36.1%, and the highest, `jni` at 330.6%. The geometric mean is 110.2%. Given the high unsafe-code execution in these benchmarks, it is not surprising to see such substantial overhead. In contrast, many benchmarks used by ERASan show little to no overhead, mainly due to their lack of unsafe code and inputs that do not sufficiently invoke unsafe code. This study highlights the effectiveness of our benchmarks.

7 Discussion: Threats to Validity

Dataset. Because of the high engineering effort required to select crates that meet our standards (Section 3.2) and the significant execution time of crates (e.g., running all crates to collect heap usage data takes several days), our dataset is limited to 100 crates. A larger dataset would yield more precise results regarding unsafe code execution.

We also expect unsafe code to be less common across *all* Rust crates than our findings suggest. Our selected crates are among the most downloaded. We counted the percentages of crates with unsafe code among (1) the top 1,000 crates (cutoff: October 2026) and (2) 1,000 randomly sampled crates updated since 2023. The results are 54.4% and 24.1%, respectively. These numbers agree with a prior study published in 2020, which found that 52.5% of the most downloaded crates and 29.4% of all crates contain unsafe code [16]. Additionally, our dataset includes only crates that contain unsafe code exercised by their workloads, while many crates do not.

Furthermore, our selection of the most downloaded crates introduces two other selection biases. First, our dataset includes only library crates and no binary crates. Second, many selected crates are low-level infrastructure libraries (e.g., `memchr`), which tend to appear deeper in application dependency graphs.

⁵Due to the rapid development of Rust, older compilers often fail to compile newer programs; the reverse also occurs, but is less common.

Workloads. The credibility of dynamic analysis greatly depends on the representativeness of the workloads. Our empirical study uses each crate’s built-in test cases and our LLM-generated drivers to call public APIs. We assess only well-maintained crates, many of which already include realistic workloads, and our LLM-generated workloads increase API coverage to 90.4%. In addition, we manually examine the workload source code to check whether the APIs are used plausibly. However, high API coverage and manual inspection do not ensure that our workloads reflect real-world usage, which also depends on API invocation frequencies and inputs—information that we generally lack.

Unsafe Instruction Identification. Tracking all instructions originating from unsafe functions or blocks across all compiler transformation passes, as discussed in Section 4.2, is impractical to implement. Instead, we identify unsafe instructions enclosed between a pair of unsafe instruction markers (§4.2). This approach is robust against most of LLVM’s optimization passes, as most optimization passes transform instructions *in place* [2]. However, optimization passes that reorder instructions may introduce both false positives—i.e., safe instructions moved into a pair of markers—and false negatives—i.e., unsafe instructions moved outside.

We investigated LLVM’s IR optimization passes [2] and found that they pose limited threats to our analysis. IR optimizations that reorder instructions are relatively uncommon and, more importantly, their code motion is usually *localized*. Additionally, our instruction markers set the `hasSideEffects` flag [1] and have no restrictive memory attributes, which prevents the markers from being optimized away and also discourages LLVM from moving memory instructions across them [34]. Therefore, unsafe instructions are likely to remain within markers, and safe instructions outside markers are unlikely to be moved inside.

For example, loop-invariant code motion (LICM), which moves loop-invariant instructions outside loops [2], is perhaps the most notable IR-reordering pass. When an unsafe block encloses a loop, LICM is unlikely to affect our analyses because (1) the markers act as instruction barriers as described above, and (2) LICM moves invariants only to nearby locations, rather than farther away. Yet, LLVM can, in principle, move instructions across marker boundaries, potentially causing our analysis to lose track of unsafe instructions or misclassify safe ones as unsafe.

Dependent Crates. We focus on assessing each crate itself to prevent unsafe code in its dependencies from dominating the results. Precisely isolating the execution time of the main crate, however, is challenging. To exclude CPU time in dependencies (§3.1.1), our pass inserts a pair of CPU counters around every call to functions in dependent crates. One challenge is that resolving indirect function calls requires precise pointer analysis, which is intractable in general [19, 46]. In our implementation, we conservatively treat all indirect function calls as targeting dependent crates. Consequently,

our measurement may overapproximate the execution time spent in dependencies, producing higher ratios of unsafe code execution for the main crate. Nevertheless, we expect the overapproximation to be minimal because indirect function calls account for only 0.35% of calls invoked in unsafe code (0.7% in the static IR). This issue can be mitigated by accompanying our analysis with approaches to indirect call resolution [12, 30, 63].

8 Related Work

Rust and its ecosystem are highly active research areas. This section focuses on prior empirical studies of unsafe Rust (§8.1) and mitigation mechanisms for unsafe Rust that currently lack effective benchmarks (§8.2). We omit other related work, such as static bug detection [8, 14, 31, 32].

8.1 Static and Manual Analysis of Unsafe Rust

Our work is closely related to prior surveys on unsafe Rust [7, 16, 44, 45, 65]. Evans et al. [16] investigated the prevalence and types of unsafe code in Rust projects, along with the reasons programmers choose to use it. Notably, they analyzed the propagation of unsafe code through call graphs, concluding that approximately 50% of Rust crates either contain unsafe code themselves or call, directly or indirectly, a function with unsafe code. Similarly, Astrauskas et al. [7] examined Rust crates to understand how programmers use unsafe Rust, focusing on whether they adhere to the best security practices—namely, using unsafe Rust sparingly and in a self-contained and well-encapsulated manner. Both Evans et al. [16] and Astrauskas et al. [7] conducted large-scale analysis of Rust crates (over 3,000) using static metrics, such as the numbers of unsafe blocks and functions. Our work complements these studies by providing insights into the execution-time characteristics of unsafe Rust, revealing discrepancies between static and dynamic unsafe code.

Qin et al. [44, 45] also investigated the reasons for using unsafe Rust. They manually analyzed instances of unsafe code usage as well as known memory safety and thread safety bugs in real-world Rust projects. Based on their findings, they proposed several programming practices to minimize the use of unsafe Rust. Similarly, Zhang et al. [66] manually inspected unsafe blocks to understand their necessity and developed a tool to auto-replace certain unsafe uses with safer alternatives.

Xu et al. [65] focused on analyzing memory safety CVEs. They summarized three categories of memory safety bugs in Rust, key challenges of avoiding them, and potential mitigation directions. While our work does not aim to detect Rust safety bugs, our dynamic analysis framework can be used to highlight unsafe code at runtime, helping developers scope potential security risks and apply mitigations strategically. For instance, developers can use runtime frequency information to decide whether to add bounds checks to raw

pointer dereferences in a hot unsafe block, depending on their overhead budget.

8.2 Unsafe Rust Mitigations

XRust [33], PKRU-Safe [26] and TRUST [9] analyze Rust on the LLVM IR level to identify memory objects accessed by unsafe code and apply intra-process memory isolation techniques to keep unsafe code from accessing memory intended exclusively for safe Rust. Similarly, MetaSafe [25] isolates and protects safety metadata of smart pointers (e.g., the length of a vector). Sandcrust [28] and Fidelius Charm [6] focus on preventing external C libraries from accessing trusted memory objects. The former isolates untrusted code in a separate process, and the latter relocates selected sensitive data into protected memory pages before running foreign functions. In addition, ERASan [42] and RustSan [13] optimize ASan [49]—a memory sanitizer originally developed to detect memory safety bugs in C/C++—for Rust by removing redundant safety checks. Similarly, LiteRSan [64] utilizes Rust-specific program analysis to identify unsafe memory accesses and enforce safety checks on them.

All these approaches introduce performance and memory overhead due to the enforcement of their security policies. They mainly rely on popular Rust programs for performance evaluation. However, many such benchmarks are inadequate for evaluating the performance and memory impact of these approaches due to a lack of unsafe code or insufficient execution time. Our benchmark suite covers a wide range of common unsafe code patterns and ensures extended execution time, making it a more effective evaluation tool for assessing unsafe Rust mitigation techniques.

9 Conclusion and Future Work

We present the first empirical study of the execution-time behavior of unsafe code in real-world Rust programs. We also curate the first Rust benchmark suite that emphasizes unsafe code for evaluating the performance overhead of Rust defenses. We show that unsafe Rust code cannot be well characterized by its static presence alone, whereas dynamic measurements provide a complementary view for understanding unsafe code and informing the development of defenses.

We see two important directions for future work. First, fine-grained dynamic analysis of unsafe code can provide more informative guidance for security analysis and tool development. For example, instead of conservatively assuming a function containing unsafe code may endanger its callers, a fine-grained analysis can track whether the callee’s unsafe code accesses resources allocated by its callers, thereby helping assess whether and how the callee’s unsafe code affects those resources. Second, more realistic benchmark workloads that reflect how the benchmarks are used in production environments can improve the evaluation of unsafe Rust defenses.

References

- [1] [n. d.]. llvml::InlineAsm Class Reference. https://llvm.org/doxygen/classllvm_1_1InlineAsm.html Accessed: 3-26-2025.
- [2] [n. d.]. LLVM's Analysis and Transform Passes. <https://llvm.org/docs/Passes.html> Accessed: 3-26-2025.
- [3] [n. d.]. Trait GlobalAlloc. <https://doc.rust-lang.org/std/alloc/trait.GlobalAlloc.html> Accessed: 04-19-2025.
- [4] 2025. The Rustonomicon: The Dark Arts of Advanced and Unsafe Rust Programming. <https://doc.rust-lang.org/nomicon/>
- [5] 2026. llvm-cov - emit coverage information. <https://llvm.org/docs/CommandGuide/llvm-cov.html>
- [6] Hussain M. J. Almhori and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA) (CODASPY '18). Association for Computing Machinery, New York, NY, USA, 248–255. <https://doi.org/10.1145/3176258.3176330>
- [7] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. <https://doi.org/10.1145/3428204>
- [8] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Tae-soo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) (SOSP '21). Association for Computing Machinery, New York, NY, USA, 84–99. <https://doi.org/10.1145/3477132.3483570>
- [9] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. 2023. TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. In *32nd USENIX Security Symposium* (USENIX Security 23). USENIX Association, Anaheim, CA, 6947–6964. <https://www.usenix.org/conference/usenixsecurity23/presentation/bang>
- [10] Sarah Bird, Aashish Phansalkar, Lizy K. John, Alex Mericas, and Rajeev Indukuru. 2007. Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture Based Processor. In *Proceedings of the SPEC Benchmark Workshop*. Austin, TX, USA. https://www.spec.org/workshops/2007/austin/papers/Performance_Characterization_SPEC_CPU_Benchmarks.pdf
- [11] Johannes Blaser, Floris Gorter, Klaus v. Gleissenthall, and Herbert Bos. 2026. QuickSafe: Targeted Hardening Against Memory Corruption. In *Proceedings of the 47th IEEE Symposium on Security and Privacy*.
- [12] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information. In *33rd USENIX Security Symposium* (USENIX Security 24). USENIX Association, Philadelphia, PA, 1383–1400. <https://www.usenix.org/conference/usenixsecurity24/presentation/cai-yuandao>
- [13] Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee. 2024. RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust. In *33rd USENIX Security Symposium* (USENIX Security 24). USENIX Association, Philadelphia, PA, 3729–3746. <https://www.usenix.org/conference/usenixsecurity24/presentation/cho-kyuwon>
- [14] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 82 (May 2023), 21 pages. <https://doi.org/10.1145/3542948>
- [15] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. 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* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 106, 13 pages. <https://doi.org/10.1145/3597503.3639136>
- [16] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 246–257. <https://doi.org/10.1145/3377811.3380413>
- [17] Rust Secure Code Working Group. [n. d.]. RUSTSEC: The Rust Security Advisory Database. <https://rustsec.org/>
- [18] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. 2005. Practical and Accurate Low-Level Pointer Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO '05). IEEE Computer Society, USA, 291–302. <https://doi.org/10.1109/CGO.2005.27>
- [19] Michael Hind. 2001. Pointer Analysis: pointer analysis or Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) (PASTE '01). Association for Computing Machinery, New York, NY, USA, 54–61. <https://doi.org/10.1145/379605.379665>
- [20] Michael Hind and Anthony Pioli. 2000. Which pointer analysis should I use?. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Portland, Oregon, USA) (ISSA '00). Association for Computing Machinery, New York, NY, USA, 113–123. <https://doi.org/10.1145/347324.348916>
- [21] Karl Huppler. 2009. The Art of Building a Good Benchmark. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–30.
- [22] Intel Corporation. 2021. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Order Number: 253665-075US.
- [23] Intranautic. 2022. glibc PTMalloc Internals. <https://intranautic.com/posts/glibc-ptmalloc-internals/> Accessed: 2025-09-25.
- [24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. <https://doi.org/10.1145/3158154>
- [25] Martin Kayondo, Inyoung Bang, Yeongjun Kwak, HyunGon Moon, and Yunheung Paek. 2024. MetaSafe: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity. In *33rd USENIX Security Symposium* (USENIX Security 24). USENIX Association, Philadelphia, PA, 3711–3728. <https://www.usenix.org/conference/usenixsecurity24/presentation/kayondo>
- [26] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) (EuroSys '22). Association for Computing Machinery, New York, NY, USA, 132–148. <https://doi.org/10.1145/3492321.3519582>
- [27] Steve Klabnik and Carol Nichols. 2022. *The Rust Programming Language*. <https://doc.rust-lang.org/stable/book/>
- [28] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (Shanghai, China) (PLOS'17). Association for Computing Machinery, New York, NY, USA, 51–57. <https://doi.org/10.1145/3144555.3144562>
- [29] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization* (CGO '04). IEEE Computer Society, Palo Alto, CA, 12 pages. <https://doi.org/10.1109/CGO.2004.1281665>
- [30] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) (PLDI '07). ACM, New York, NY, USA, 278–289. <https://doi.org/10.1145/1250734.1250766>

- [31] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, Republic of Korea) (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2183–2196. <https://doi.org/10.1145/3460120.3484541>
- [32] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 2022. Detecting Cross-language Memory Management Issues in Rust. In *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III (Copenhagen, Denmark)*. Springer-Verlag, Berlin, Heidelberg, 680–700. https://doi.org/10.1007/978-3-031-17143-7_33
- [33] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing Unsafe Rust Programs with X Rust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 234–245. <https://doi.org/10.1145/3377811.3380325>
- [34] LLVM [n. d.]. *LLVM Language Reference Manual*. <https://llvm.org/docs/LangRef.html> <https://llvm.org/docs/LangRef.html>
- [35] LLVM Project. 2024. Opaque Pointers. <https://releases.llvm.org/18.1.1/docs/OpaquePointers.html> LLVM 18.1.1 documentation.
- [36] Matt Mastracci. [n. d.]. dtor. <https://crates.io/crates/dtor>
- [37] Niko Matsakis. 2016. Introducing MIR. <https://blog.rust-lang.org/2016/04/19/MIR.html> Accessed: 09-24-2022.
- [38] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (Portland, Oregon, USA) (HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. <https://doi.org/10.1145/2663171.2663188>
- [39] Max Planck Institute for Software Systems. [n. d.]. RustBelt: Logical Foundations for the Future of Safe Systems Programming. <https://plv.mpi-sws.org/rustbelt/>. Accessed: 2026-06-08.
- [40] Alice Maz and Marshall Pierce. [n. d.]. base64. <https://crates.io/crates/base64>
- [41] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS, Vol. 22*. 1–17.
- [42] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. ERASan: Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4053–4068. <https://doi.org/10.1109/SP54263.2024.00258>
- [43] The Rust Project. [n. d.]. Rust Compiler Development Guide. <https://rustc-dev-guide.rust-lang.org/> Accessed: 2/14/2025.
- [44] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. 2024. Understanding and Detecting Real-World Safety Issues in Rust. *IEEE Trans. Softw. Eng.* 50, 6 (March 2024), 1306–1324. <https://doi.org/10.1109/TSE.2024.3380393>
- [45] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. 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 (London, UK) (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. <https://doi.org/10.1145/3385412.3386036>
- [46] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. <http://www.jstor.org/stable/1990888>
- [47] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Annual Computer Security Applications Conference (Virtual Event, USA) (ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 824–836. <https://doi.org/10.1145/3485832.3485903>
- [48] Secure Code Working Group. 2023. RustSec Advisory Database. <https://rustsec.org/>. Accessed: 2023.
- [49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>
- [50] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for Embedded Systems: Current State and Open Problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (Salt Lake City, UT, USA) (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2296–2310. <https://doi.org/10.1145/3658644.3690275>
- [51] Sarabjeet Singh and Manu Awasthi. 2019. Memory Centric Characterization and Analysis of SPEC CPU2017 Suite. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (Mumbai, India) (ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 285–292. <https://doi.org/10.1145/3297663.3310311>
- [52] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (Barcelona, Spain) (CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. <https://doi.org/10.1145/2892208.2892235>
- [53] The Rust Project Developers. 2026. *The rustc Book*. The Rust Project. <https://doc.rust-lang.org/rustc/> Accessed: 2026-05-28.
- [54] The Rust Team. 2025. cargo-bench. <https://doc.rust-lang.org/cargo/commands/cargo-bench.html>. Accessed: 2025-09-23.
- [55] The Rust Team. 2025. Cargo.toml vs Cargo.lock. <https://doc.rust-lang.org/cargo/guide/cargo-toml-vs-cargo-lock.html> Accessed: 2025-09-24.
- [56] The Rust Team. 2026. cargo-doc. <https://doc.rust-lang.org/cargo/commands/cargo-doc.html>. Accessed: 2026-05-30.
- [57] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (Austin, Texas, USA) (ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 333–336. <https://doi.org/10.1145/2668930.2688819>
- [58] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High System-Code Security with Low Overhead. In *2015 IEEE Symposium on Security and Privacy*. 866–879. <https://doi.org/10.1109/SP.2015.58>
- [59] The Rust Security Response WG. 2022. Security advisory for the standard library (CVE-2022-21658). <https://blog.rust-lang.org/2022/01/20/cve-2022-21658/>
- [60] The Rust Security Response WG. 2024. Security advisory for the standard library (CVE-2024-24576). <https://blog.rust-lang.org/2024/04/09/cve-2024-24576/>
- [61] The Rust Security Response WG. 2024. Security advisory for the standard library (CVE-2024-43402). <https://blog.rust-lang.org/2024/09/04/cve-2024-43402/>
- [62] Secure Code working group. 2026. RustSec Advisory Database. <https://rustsec.org/>
- [63] Tianrou Xia, Hong Hu, and Dinghao Wu. 2024. DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5877–5894. <https://www.usenix.org/conference/usenixsecurity24/presentation/xia>
- [64] Tianrou Xia, Kaiming Huang, Dongyeon Yu, Yuseok Jeon, Jie Zhou, Dinghao Wu, and Taegyu Kim. 2025. LiteRSan: Lightweight Memory Safety Via Rust-specific Program Analysis and Selective Instrumentation. arXiv:2509.16389 [cs.CR] <https://arxiv.org/abs/2509.16389>
- [65] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* 31, 1,

Article 3 (sep 2021), 25 pages. <https://doi.org/10.1145/3466642>

- [66] Yuchen Zhang, Ashish Kundu, Georgios Portokalidis, and Jun Xu. 2023. On the Dual Nature of Necessity in Use of Rust Unsafe Code. In *Proceedings of the 31st ACM Joint European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) (*ESEC/FSE 2023*). Association for Computing Machinery, New York, NY, USA, 2032–2037. <https://doi.org/10.1145/3611643.3613878>