Miri: Practical Undefined Behavior Detection for Rust

RALF JUNG, ETH Zurich, Switzerland
BENJAMIN KIMOCK, Lansweeper NV, USA
CHRISTIAN POVEDA, Unaffiliated, Colombia
EDUARDO SÁNCHEZ MUÑOZ, Unaffiliated, Spain
OLI SCHERER, Unaffiliated, Germany
QIAN WANG, Unaffiliated, UK

The Rust programming language has two faces: on the one hand, it is a high-level language with a strong type system ensuring memory and thread safety. On the other hand, Rust crucially relies on *unsafe code* for cases where the compiler is unable to statically ensure basic safety properties. The challenges of writing unsafe Rust are similar to those of writing C or C++: a single mistake in the program can lead to *Undefined Behavior*, which means the program is no longer described by the language's Abstract Machine and can go wrong in arbitrary ways, often causing security issues.

Ensuring the absence of Undefined Behavior bugs is therefore a high priority for unsafe Rust authors. In this paper we present *Miri*, the first tool that can find *all* de-facto Undefined Behavior in deterministic Rust programs. Some of the key non-trivial features of Miri include tracking of pointer provenance, validation of Rust type invariants, data-race detection, exploration of weak memory behaviors, and implementing enough basic OS APIs (such as file system access and concurrency primitives) to be able to run unchanged real-world Rust code. In an evaluation on more than 100 000 Rust libraries, Miri was able to successfully execute more than 70% of the tests across their combined test suites. Miri has found dozens of real-world bugs and has been integrated into the continuous integration of the Rust standard library and many prominent Rust libraries, preventing many more bugs from ever entering these codebases.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; Semantics.

Additional Key Words and Phrases: Rust, bug finding, Undefined Behavior, Miri

ACM Reference Format:
Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL, Article 48 (January 2026), 29 pages. https://doi.org/10.1145/3776690

1 Introduction

Rust [39] is primarily known for being a *safe* systems programming language: thanks to a strong type system, programmers have control over low-level concerns such as the placement of data in memory while still enjoying high-level memory safety and thread safety guarantees [15]. However, sometimes the Rust type system is not strong enough to verify that a piece of code is actually safe. In that case, programmers can reach for *unsafe code*. Marked with an explicit `unsafe` keyword, such code is permitted to call operations where memory safety guarantees must be upheld by the caller. A typical example is an unchecked array access: to squeeze the maximum performance out

Authors' Contact Information: Ralf Jung, ralf.jung@inf.ethz.ch, ETH Zurich, Department of Computer Science, Zurich, Switzerland; Benjamin Kimock, kimockb@gmail.com, Lansweeper NV, USA; Christian Poveda, contact@pvdrz.com, Unaffiliated, Colombia; Eduardo Sánchez Muñoz, eduardosm-dev@e64.io, Unaffiliated, Spain; Oli Scherer, rust@oli-obk.de, Unaffiliated, Germany; Qian Wang, andy.wang99@icloud.com, Unaffiliated, UK.

ACM 2475-1421/2026/1-ART48
https://doi.org/10.1145/3776690

of a hot loop, the programmer can promise to the compiler that an array access will definitely always be in-bounds, and then the compiler will omit the bounds check.

However, with great power comes great responsibility. If the programmer gets it wrong, and the code is not *actually* safe, the program can misbehave in basically arbitrary ways. To illustrate that, consider the following example:

Example 1

```
let x: i8 = unsafe { MaybeUninit::uninit().assume_init() };
assert!(x < 0 || x == 0 || x > 0);
println!("{x}");
```

This (very much non-representative) piece of Rust code first constructs a signed 8-bit integer by creating an uninitialized chunk of memory and then (incorrectly!) claiming to the compiler that this memory has been initialized by calling `assume_init`. Then, it checks whether the resulting integer satisfies a simple tautology. The final line of the program prints the value of the integer. Surprisingly, if we build this program with version 1.88 of the Rust compiler (with optimizations enabled), the output is: `assertion failed: x < 0 || x == 0 || x > 0`. Apparently, the compiler came up with an integer that does not satisfy this property?

This is a typical example of *Undefined Behavior* (UB): if the program violates the compiler-assumed preconditions of an operation like `assume_init`, then the behavior of the compiled program may be entirely arbitrary. Undefined Behavior is notoriously unstable; for instance, moving the `println!` in the example one line up makes the error disappear (the program just prints `0` then), and removing it entirely leads to a completely different error message. This makes Undefined Behavior a debugging nightmare: standard source-level reasoning principles simply do not apply anymore.

Furthermore, Undefined Behavior is a massive security problem. Around 70% of critical security vulnerabilities are caused by memory safety violations [38, 18, 32], and all of these memory safety violations are instances of Undefined Behavior. After all, if the attacker overflows a buffer to eventually execute their own code, this is not something that the program does because the C or C++ specification says so—the specification just says that doing out-of-bounds writes (or overwriting the vtable, or calling a function pointer that does not actually point to a function, or doing any of the other typical first steps of an exploit chain) is Undefined Behavior, and executing the attacker's code is just how Undefined Behavior happens to play out in this particular case.

It is, therefore, highly desirable to ensure that a program is free of UB. In many languages, this property is ensured by the type system and/or the language runtime: the well-known slogan for type soundness could be (less eloquently) rephrased as "well-typed programs don't have Undefined Behavior". However, not all code can be written in such languages—when code is too tricky to automatically rule out Undefined Behavior, programmers reach for languages such as C, C++, or unsafe Rust to achieve their goals. How can we help those programmers avoid UB in their code?

For cases like this, fully ruling out Undefined Behavior in all possible executions is often out of the question—this requires full-scale formal verification tools, along with significant expertise and effort. Improving the practicality and efficiency of such tools is the subject of a lot of recent and ongoing work; however, in this paper, we focus on a different approach: *dynamically testing for Undefined Behavior*. Instead of asking the tool to answer the (undecidable) question of whether *any* execution can have UB, we consider the problem of checking whether a *single* execution has UB. Programmers can then, for instance, apply this check to their code's test suite—assuming good test coverage, this can ensure that many common ways of interacting with their code avoid UB.

The most widely-used approach for performing Undefined Behavior testing is to use a form of instrumentation called a *sanitizer*. By compiling the program with extra assertions (for instance, checking that every memory access is within the bounds of an allocation), many cases of Undefined Behavior can be detected at runtime. However, while sanitizers are able to detect some cases of

Undefined Behavior, no current sanitizer is able to rule out all UB, not even for a single execution. The main reason for this is that UB checking can require extra state not available to the sanitizer, such as the provenance [29] of a pointer. Furthermore, some UB (such as sequence point violations in C) may simply not be represented in the program any more when code reaches the stage during compilation at which the instrumentation occurs.

In this paper we present *Miri*, the first tool that can find *all* the de-facto Undefined Behavior in deterministic Rust programs.[1] Miri achieves this by taking a formal-methods-inspired approach: at the heart of Miri is a direct implementation of an Abstract Machine for Rust. However, despite the overhead that comes with executing code on the Abstract Machine rather than directly on the CPU, Miri is highly practical: distributed as part of the official "nightly" releases of Rust, many of the core libraries of the Rust ecosystem use Miri as part of their development process to rule out Undefined Behavior. Miri has found dozens of bugs in the Rust standard library and other widely-used libraries, and likely prevented many more bugs from ever entering these libraries by detecting them before the faulty code got merged. 70% of the tests in the 100 000 most widely-used Rust libraries can be successfully executed with Miri.

The rest of this paper is structured as follows: First, we showcase Miri's main features by demonstrating how it detects Undefined Behavior in a series of small examples (§2). Then, we describe an idealized form of Miri's core architecture by explaining all the state that Miri tracks during program execution (§3). In §4, we explain in more detail how Miri deals with concurrent programs; specifically, we explain Miri's data race detector and its ability to exhibit weak memory effects. We continue in §5 by explaining the key optimizations we applied to the previously described core architecture, and other aspects of Miri that helped making it a practical tool. Next, we describe our evaluation of Miri on a large fraction of the open-source Rust ecosystem, we report on Miri's performance, and we briefly describe some of the real-world bugs Miri was able to find (§6). Finally, we discuss related work (§7).

2 Showcasing Miri

Before we dig deeper into Miri's architecture and abilities, we want to set the scene by giving a brief overview of the kind of bugs Miri is able to catch.

Uninitialized memory. We start with Example 1 from the introduction. Miri says:

```
error: Undefined Behavior: using uninitialized data, but this operation requires initialized memory
 --> src/main.rs:4:26
  |
4 |     let x: i8 = unsafe { MaybeUninit::uninit().assume_init() };
  |                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Undefined Behavior occurred here
```

The error indicates that the problem in fact lies in the very first line of the program: As the documentation for `MaybeUninit` would explain, Rust fundamentally distinguishes between *initialized* and *uninitialized* memory. While any bit pattern one can actually observe on a real computer could be interpreted as an 8-bit signed integer,[2] the Rust documentation says that uninitialized memory is an "abstract" state distinct from all these "concrete" states, and that it is invalid to interpret uninitialized memory as an integer.

The way this works is that at its heart, Miri is an *interpreter*. Instead of compiling Rust code to machine code, it takes one of the intermediate representations of the Rust compiler (MIR, the "middle" IR) and interprets it statement by statement, basic block by basic block. This gives Miri the freedom to track all sorts of additional data that would not be present in a normal execution of the

[1]See §3.4 for why we have to qualify this claim by talking about *de-facto* Undefined Behavior and *deterministic* programs.
[2]At least, on all contemporary machines. Itanium CPUs had a distinct state indicating an uninitialized register.

compiled program—such as which exact parts of memory are initialized. We will describe the full extent of that additional state in §3; for now, we continue with our example-based tour of Miri.

Temporal and spatial memory safety violations. Miri is able to detect out-of-bounds and use-after-free accesses. For instance, this Rust code has a use-after-free bug:

Example 2

```
let mut b = Box::new(0); // Allocate some memory on the heap.
let ptr = &raw mut *b; // Create a "raw" pointer to that memory.
drop(b); // Free the heap allocation.
unsafe { *ptr = 1 }; // Write to the freed memory.
```

Miri shows the following detailed error message:

```
error: Undefined Behavior: memory access failed: alloc289 has been freed, so this pointer is dangling
 --> src/main.rs:7:14
  |
7 |     unsafe { *ptr = 1 }; // Write to the freed memory.
  |              ^^^^^^^^ Undefined Behavior occurred here
help: alloc289 was allocated here:
 --> src/main.rs:4:17
  |
4 |     let mut b = Box::new(0); // Allocate some memory on the heap.
  |                 ^^^^^^^^^^^
help: alloc289 was deallocated here:
 --> src/main.rs:6:5
  |
6 |     drop(b); // Free the heap allocation.
  |     ^^^^^^^
```

This time, the actual detection of the error is not that complicated: at every memory access, we can simply check whether the pointer used for that access still points to valid memory. However, to provide good error messages, we also track where in the program every piece of memory was allocated and where it has been deallocated. To avoid accumulating information about irrelevant allocations, Miri has a GC that periodically scans for allocations for which no pointer exists anymore; those can then be entirely purged from the state.

Memory leaks. This is the one kind of bug Miri can catch that is not actually Undefined Behavior: when the program finishes executing, we check whether there is still any memory around that has not been properly freed. If that is the case, we emit an error like this:

```
error: memory leaked: alloc289 (Rust heap, size: 4, align: 4), allocated here:
 --> src/main.rs:2:13
  |
2 |     let b = Box::new(42);
  |             ^^^^^^^^^^^^
```

Data races. Miri includes a data race checker based on vector clocks (see §4), which allows it to detect unsynchronized memory accesses. For instance, this code incorrectly tries to read and write `GLOBAL` in two threads “at the same time”, *i.e.,* without synchronization:

Example 3

```
static mut GLOBAL: i32 = 0;
thread::spawn(|| unsafe { GLOBAL = 1 });
thread::spawn(|| println!("GLOBAL={}", unsafe { GLOBAL }));
```

Just like in C and C++, it is invalid for two threads in a Rust program to access the same data without proper synchronization, *e.g.,* through a lock (except if both accesses are reads).

When Miri runs this program, it can *almost always* find this bug: as is typical for concurrent programs, the above example is non-deterministic, and therefore has many possible executions. If the first thread finishes before the second thread even gets spawned, and if furthermore memory allocator interactions lead to synchronization between the first thread and the main thread, then that particular execution does not have a data race. Miri uses a pseudo-random number generator with a fixed seed to resolve non-deterministic choices. and it supports a mode where the program is executed with many different seeds concurrently. In this mode, Miri finds this data race for around 90% of the executions.

The error message reported by Miri points at both conflicting accesses:

```
error: Undefined Behavior: Data race detected between (1) non-atomic write on thread `unnamed-1`
and (2) non-atomic read on thread `unnamed-2` at alloc1
 --> src/main.rs:8:51
  |
8 |     thread::spawn(|| println!("GLOBAL={}", unsafe { GLOBAL }));
  |                                                     ^^^^^^ (2) just happened here
help: and (1) occurred earlier here
 --> src/main.rs:7:29
  |
7 |     thread::spawn(|| unsafe { GLOBAL = 1 });
  |                               ^^^^^^^^^^
```

Weak memory behaviors. Miri's support for detecting concurrency bugs goes further than data races—however, here we start really hitting the inherent limitations of Miri as a *testing* tool. For instance, consider this program:

Example 4

```
static X: AtomicI32 = AtomicI32::new(0);
static Y: AtomicI32 = AtomicI32::new(0);
thread::scope(|s| {
    s.spawn(|| { X.store(1, Ordering::Relaxed); Y.store(1, Ordering::Relaxed); });
    s.spawn(|| { if Y.load(Ordering::Relaxed) == 1 {
        assert!(X.load(Ordering::Relaxed) == 1);
    } });
});
```

Here, we can see Rust's *atomics* in action. Their general principle is inherited directly from C++: while concurrent regular reads and writes to the same memory by different threads are disallowed (as demonstrated by the previous example), Rust programmers can use atomic types to explicitly mark certain memory accesses as being "potentially racy". Such accesses *are* allowed to occur without proper synchronization. However, the naive semantics of "programs behave as-if threads take turns one after the other" turns out to be quite expensive to implement on real-world hardware, so C++ introduced a *weak memory model* [3] that allows programmers to mark operations with the requested *memory ordering*. Weaker orderings are more efficient, but also allow more "strange" program behaviors. The example above uses relaxed atomic accesses, which is the weakest ordering.

This program works on two global variables, X and Y. The first thread first stores 1 into X and then does the same for Y. The second thread first reads Y, and if it sees the value 1, it concludes that the first thread is done, and therefore X must also be 1 now.

Unfortunately, this argument is flawed: relaxed accesses do not have to be observed in a consistent order by all threads. Just because the first thread wrote to X before Y, does not mean that the second thread will see the two writes in that order. In practice, they can be reordered by compiler optimizations, or by being cached in the CPU and then being flushed out-of-order. Either of these effects can lead to the second thread observing a value of 1 for Y *before* the store of X propagates to this thread, thus causing an assertion failure.

As with the previous example, this program is non-deterministic, so Miri can only *sometimes* find this bug. When running Miri with many different seeds, we encounter the assertion failure in around 25% of the executions.

Rust type invariants. The Rust compiler expects all variables to uphold certain basic invariants given by their type. For instance, a value of type `bool` must always be `0x00` or `0x01`. This is exploited by optimizations (providing information about the possible ranges of values in the program), and is also exploited by the logic that decides how types are laid out in memory (for instance, `Option<bool>` can be stored in a single byte).

Of course, in entirely safe code this property trivially holds, but unsafe code can easily accidentally violate these invariants when using low-level programming patterns:

Example 5

```
struct S { f1: bool, f2: i16 }
let mut s: S = unsafe { mem::transmute(u32::MAX) };
```

This program works with a type `S` that has two fields: a `bool` and a two-byte singed integer. It then uses `mem::transmute` to do a type-unsafe conversion of the four-byte integer `0xFFFFFFFF` to `S`. This initializes `f1` with the value `0xFF`, which is invalid for `bool`:

```
error: Undefined Behavior: constructing invalid value at .f1: encountered 0xff, but expected a boolean
 --> src/main.rs:4:25
  |
4 | let mut s: S = unsafe { mem::transmute(u32::MAX) };
  |                         ^^^^^^^^^^^^^^^^^^^^^^^^ Undefined Behavior occurred here
```

Note that the error says which exact field of `S` had its invariant violated.

Pointer provenance. The last example we consider demonstrates one of the most subtle issues that can arise in unsafe Rust: incorrect pointer provenance [29]. The code demonstrating this is a bit more complicated, so we will go over it slowly: (we omit the surrounding `unsafe` block)

Example 6

```
let mut x = 0u8; let xptr = &raw mut x;
let mut y = 1u8; let yptr = &raw mut y;
// Make a second pointer to x
let x_minus_y = (xptr as usize).wrapping_sub(yptr as usize);
let xptr2 = yptr.wrapping_add(x_minus_y); // y + (x-y)

// Use both pointers
xptr2.write(2);
assert!(xptr.read() == 2);
```

The first two lines set up two local variables, `x` and `y`, and a pointer to each of them. The next two lines use pointer-to-integer casts to compute `y + (x-y)` and store the result in `xptr2`. At this point, `xptr` and `xptr2` are "equal" in the sense that `xptr == xptr2` would evaluate to `true` (but as we will soon see, there is still a crucial difference between those two pointers). Finally, we write through `xptr2` and then read from `xptr`, and expect to read back the value we just wrote.

This may look like a perfectly innocent, if somewhat odd, use of pointer arithmetic. However, if we build this code with optimizations with Rust 1.88, curiously, the assertion fails. It turns out that this program has Undefined Behavior, and Miri confirms this:

```
error: Undefined Behavior: memory access failed: attempting to access 1 byte, but got alloc246-0x9
which points to before the beginning of the allocation
  --> <source>:12:9
   |
12 |     xptr2.write(2);
   |     ^^^^^^^^^^^^^^ Undefined Behavior occurred here
help: alloc246 was allocated here:
  --> <source>:3:9
   |
3  |     let mut y = 1u8; let yptr = &raw mut y;
   |         ^^^^^
```

What is happening here? We can get a clue by reading the documentation for `wrapping_add`:

> The resulting pointer "remembers" the allocation that `self` points to; it must not be used to read or write other allocations.

The optimizer, when analyzing this code, realizes that the last line reads from `x` and will attempt to determine whether any of the previous instructions could have mutated `x`. It comes to the conclusion that writing through `xptr2` can *not* mutate `x` because `xptr2` is "derived from" `y` via `yptr`, *i.e.*, it was computed in a way that the resulting pointer is only allowed to point to `y`, not to any other allocation. Therefore, the compiler deduces that the value of `x` cannot have changed until the final assertion, so `xptr.read()` must return `0`.

Miri detects this problem, as the documentation suggests, by "remembering" where the pointer comes from: the allocation backing `y` (numbered `alloc246`). The `alloc246-0x9` part of the error indicates that the pointer has been offset 9 bytes to the left of where `y` is located. The fact that `x` is stored at that same address is irrelevant: every pointer has exactly one allocation that it may access, and using that pointer to do a read or write outside the bounds of that allocation is UB, even if that access happens to be inbounds of another allocation. This extra information attached to a pointer is called *provenance*. It is erased during regular compilation, which is why `xptr == xptr2` would return `true`, but it is highly relevant when determining whether a program has UB or not.

Aliasing violations. There is one more class of UB in Rust: violating the aliasing requirements of Rust's reference types. However, equipping Miri with the ability to check these requirements has been the subject of prior work [14, 42], so we do not discuss it as part of this paper.

3 Miri Architecture Overview

Having an idea of the kind of bugs Miri is able to find, we now move on to give an overview of Miri's architecture. This is highly idealized in the sense that it is conceptually the architecture we need for Miri to do its job, but a direct implementation of this architecture would be prohibitively slow. In §5, we will discuss how Miri refines this architecture to achieve practical performance. We also simplify some aspects of the interpreter for reasons of space, *e.g.*, we omit everything that is related to supporting Rust's dynamically-sized types (*i.e.*, slices and trait objects), and we omit state that is tracked solely for the purpose of improved error messages.

As already mentioned, Miri is an *interpreter*. Miri is tightly integrated with the Rust compiler, reusing the entire frontend and middle-end.[3] Specifically, Miri is based on the MIR stage of the compiler, which is usually the last stage before generating LLVM IR. This is also where the name comes from: **MIR I**nterpreter. MIR is a convenient choice since at this point, all control flow has been compiled to a simple control-flow graph, and most complicated Rust operations have been desugared into simple assignments and function calls. In particular, pattern matching has been compiled into plain conditional jumps, saving us the effort of having to entirely re-implement Rust's non-trivial (and evolving) pattern matching semantics. At the same time, MIR is still fully typed, giving Miri all the information it needs to detect Undefined Behavior, including enforcing Rust type invariants. We turn off all MIR optimizations as those may remove UB.

Aside from the language we are interpreting, the other fundamental decision we have to make is to choose how the runtime state of the program should be represented. As the examples in the previous section show, it is not sufficient to just imitate what would happen in the compiled form of the program: we need to additionally track which parts of memory have been initialized, we need to track the provenance [29] of each pointer, and we need to track meta-data for the data race

[3]In fact, Miri is so tightly integrated that parts of it live inside the compiler: the core interpreter (and most of the state we describe in this section) is shared with the Rust compiler's compile-time function evaluation infrastructure. This leads to some rifts in the architecture, separating Miri-specific state from state that is also present in the shared core. For purposes of presentation in this paper, we present Miri as-if there was no such separation.

$$\begin{aligned} \mathit{Pointer} &\triangleq \{\ \mathsf{addr} : \mathbb{N}, \mathsf{prov} : \mathit{Provenance}\ \} & \mathit{Scalar} &\triangleq \mathsf{Int}(\mathbb{Z}) \mid \mathsf{Ptr}(\mathit{Pointer}) \\ \mathit{Provenance} &\triangleq \mathsf{Known}(\mathit{AllocId}) \mid \mathsf{Wildcard} & \mathit{Operand} &\triangleq \mathsf{Scalar}(\mathit{Type} \times \mathit{Scalar}) \\ & & &\quad \mid \mathsf{NonScalar}(\mathit{Type} \times \mathit{List}(\mathit{Byte})) \end{aligned}$$

Fig. 1. Values in Miri

detector. Furthermore, we need to represent the fact that memory is structured: when explaining the pointer provenance example (Example 6), we used the concept of an “allocation”, referring to the fact that memory is partitioned in blocks and the borders between those blocks matter to determine which pointer is allowed to access which memory. And finally, memory does not just contain allocations that consist of bytes; we use memory as a term for “everything a pointer can point to”, and in Rust, that also includes functions.[4] All of this means that our memory is quite far from the simple idea that “memory is just a huge array of bytes”.

3.1 Values and operands

Before we can explain the representation of memory in Miri, we have to briefly talk about what kinds of *values* Miri manipulates—and what it even uses values for. Unlike in a typical interpreter for strongly typed languages, values do *not* show up anywhere in Miri’s state. This is because Rust allows unsafe code to manipulate memory on a per-byte level, completely bypassing any high-level view of memory. However, values still play an important role in Miri transiently, *i.e.,* during the execution of a single instruction: the typical operation will load some values from memory, operate on those values to compute new values, and then store those new values back into memory.

A simplified overview of Miri’s value hierarchy is presented in Figure 1. To align better with MIR terminology, a general value is called *operand* (since they only appear as arguments to an operation). Operands carry their type with them; this is crucial because the same mathematical integer may look very different in memory depending on the type it is stored at. Operands broadly fall into two categories: those that have a high-level, mathematical representation (*scalars*), and those that are just represented by their raw list of bytes. The only operations on non-scalars are loading them from somewhere in memory, and storing them to somewhere else in memory, so we do not need any more structure than a list of bytes. (However, we have to be careful: copying operands does not always exactly preserve all the bytes. The padding between fields should not be preserved on a copy, and pointer provenance is not always perfectly preserved either. Miri post-processes the result of a copy to erase any data that should not have been preserved.)

For all other operations where more structure is needed, operands need to be converted to scalars. Those come in two forms: (mathematical) integers and pointers. Pointers are unusual in that they are *not* just an absolute address indicating where the pointer points to. As demonstrated by Example 6, Miri needs to sometimes distinguish two pointers to the same address based on which other pointers they have been derived from. Concretely, we do this by enriching the pointer with *provenance*. Typically, provenance takes the form of an *allocation ID*: each time an allocation (such as a local variable or a heap allocation) gets created, the initial pointer is equipped with a unique ID representing that allocation; that ID is preserved by operations such as pointer arithmetic so that we can always identify which allocation a pointer “belongs to”. However, there are cases where the concrete provenance of a pointer is unknown: specifically, Rust supports casting pointers

[4]It also includes “vtables”, an implementation detail of `dyn Trait` types which are Rust’s approach to run-time polymorphism and dynamic dispatch. Miri fully supports vtables, but we will gloss over them in this paper.

$$
\begin{aligned}
\mathit{AllocId} &\triangleq \mathbb{N} \\
\mathit{Memory} &\triangleq \{\ \mathsf{allocs} : \mathit{AllocId} \xrightarrow{\text{fin}} \mathbb{N} \times \mathit{Alloc}, \\
&\qquad \mathsf{exposed} : \mathcal{P}(\mathit{AllocId})\ \} \\
\mathit{Alloc} &\triangleq \mathsf{DataAlloc}(\mathit{DataAlloc}) \\
&\quad \mid \mathsf{FnAlloc}(\mathit{Instance})
\end{aligned}
\qquad
\begin{aligned}
\mathit{DataAlloc} &\triangleq \{\ \mathsf{data} : \mathit{List}(\mathit{Byte} \times \mathit{DataRaceLoc}), \\
&\qquad \mathsf{mutable} : \mathbb{B}, \\
&\qquad \mathsf{align} : \mathbb{N}\ \} \\
\mathit{Byte} &\triangleq \mathsf{Uninit} \\
&\quad \mid \mathsf{Init}(\mathsf{val} : \mathbb{N}_{<256}, \mathsf{prov} : \mathit{Provenance}_\perp)
\end{aligned}
$$

Fig. 2. The representation of memory in Miri

to integers and back; when that happens, Miri marks the resulting pointer as a Wildcard pointer. We will get back to this at the end of the next subsection.

3.2 Memory

Figure 2 shows the structure of memory in Miri. Miri's memory follows a typical block-based model in the style of CompCert [25]. Memory consists of *allocations* that are identified by an ID (which already came up for *Provenance* in the previous subsection). The *allocs* field stores the absolute address and the contents of each allocation. An allocation can either contain data or it can contain a function. A function is identified by an Instance, which is a Rust compiler data type; the details of this do not matter much.[5] Moving on to the typical case of an allocation containing data; those allocations store the list of bytes they contain, whether the allocation is mutable, and the *alignment* the allocation was allocated with.[6] Alongside each byte in the allocation, we also have some metadata for data race detection. And finally, each byte in the allocation explicitly tracks whether it is initialized, and initialized bytes optionally can carry a *provenance*. (The subscript $\perp$ indicates an option/maybe-type: $T_\perp \triangleq T \mid \perp$.) Without this prov field in a *Byte*, there would be no way to store a *Pointer* (§3.1) in memory and later restore the exact original value.

For example, here's an (abbreviated) memory that contains two allocations: the first allocation (with ID 0) at address 16 is 4 bytes large, with the first two bytes being initialized to 0 and the last two bytes being uninitialized; the second allocation (with ID 1) at address 32 holds a pointer to the first allocation. As a scalar, that pointer would be written $\mathsf{Ptr}(\{\ \mathsf{addr} : 16, \mathsf{prov} : \mathsf{Known}(0)\ \})$.

Example 7

$$
\begin{aligned}
&0 \mapsto (16, \mathsf{DataAlloc}(\{\ \mathsf{data} : [\mathsf{Init}(0,\perp), \mathsf{Init}(0,\perp), \mathsf{Uninit}, \mathsf{Uninit}]\ \})) \\
&1 \mapsto (32, \mathsf{DataAlloc}(\{\ \mathsf{data} : [\mathsf{Init}(16, \mathsf{Known}(0)), \mathsf{Init}(0, \mathsf{Known}(0)), \ldots, \mathsf{Init}(0, \mathsf{Known}(0))]\ \}))
\end{aligned}
$$

Note how the pointer is represented by 8 bytes that each carry same provenance, and whose values encode the absolute address of the memory they point to (in little-endian encoding, *i.e.,* with the least significant byte first).[7] Meanwhile, the bytes in allocation 0 do *not* carry provenance since they do not represent a pointer.

Exposed provenance and integer-pointer casts. The one field we have not discussed yet is *Memory*.exposed. This field has to do with pointer-integer casts: Miri implements a variant of the PNVI model [29] that has been proposed for C, specifically a mix of PNVI-escaped (also known

[5]It is called an "instance" since Rust functions can be generic, and here we refer to a particular instance of a function, *i.e.,* a particular choice for all the generic parameters.

[6]An alignment of n means that the address of the allocation must be divisible by n. We need to remember the originally requested alignment for each allocation since some allocator APIs require the programmer to indicate the original size and alignment of an allocation upon deallocation, which simplifies the internal bookkeeping of the allocator. The *align* field allows Miri to detect cases where the programmer provides the wrong information upon deallocation.

[7]Miri is parametric in the pointer size and endianess; here we use the by far most common 64-bit little-endian configuration.

as PNVI-address-exposed / PNVI-ae) and PNVI-wildcard. This means that every time a pointer is cast to an integer, the allocation it is derived from (as determined by its provenance) is marked as *exposed* by adding it to the exposed set. Similar to PNVI-wildcard, when an integer is cast to a pointer, we do not try to make a best guess about which allocation this pointer should be associated with—we just give it a Wildcard provenance. When a pointer with that provenance is used to access memory, we check whether some *exposed* allocation exists at the pointer's address, and reject the memory access if not. This restriction to only permit accesses to exposed allocations makes the model different from PNVI-wildcard.

This means we will accept programs that use the same pointer to access different exposed allocations, which is definitely not desirable for a language semantics, and hence rejected by PNVI-ae and the variant PNVI-ae-udi [11] that eventually got adopted in a Technical Report by the C standards committee. Unfortunately, PNVI-ae is too limiting (it rejects programs that we would like to accept), and it is unclear how to combine PNVI-ae-udi with Miri's full model of provenance that also includes support for Stacked Borrows [14] and Tree Borrows [42]. The "user-disambiguation" (udi) mechanism is not compatible with models where more than one provenance could be valid to use for any given memory access, and it matters *which* provenance is used.

So, instead, we decided that programs that cast integers to pointers are not considered fully supported by Miri, and we warn the user about this when such a cast happens. The wildcard mechanism means we might miss some bugs in such programs, but Miri is still useful for finding other bugs, and Miri will not report false bugs.

3.3 Stack and threads

We move on to discussing the representation of the call stack—or rather, the call stack**s** and associated data for each thread, since Miri can interpret concurrent programs. The relevant definitions can be found in Figure 3. We begin by going over the parts that are relevant for a regular sequential execution, then we discuss the extra state needed to handle *unwinding* and *concurrency*.

The core components of each frame are the body (*i.e.*, the source code) of the function being executed and the location inside the function that will be executed next. MIR is a control-flow-graph-based language, so a function body consists of a bunch of basic blocks (identified by a *BlockId*) that each contain a list of statements followed by a terminator (such as a function call or a conditional).[8] Therefore, the loc field identifies the next instruction via the index of the currently executed basic block, and the index of the statement/terminator inside the basic block. The field locals stores where in memory the contents of the function's local variables are stored. Note that Rust supports creating pointers that point to local variables; the common representation of local variables as mapping names to their current *value* therefore would not work. The value needs to be stored in memory. Finally, return_place stores where the return value of the function should be stored, and return_cont stores the basic block in the *caller* that execution should continue at.

Unwinding. However, returning with a value is not the only way execution can leave a Rust function. Rust has a *panic* mechanism triggered by fatal bugs such as an out-of-bounds array access. By default, a panic will cause the program to *unwind* its stack, similar to what happens when an exception is thrown in other languages. This gets complicated by the fact that Rust, similar to C++, follows the RAII pattern (Resource Acquisition Is Initialization): every local variable has an associated destructor, and upon panicking, the destructors of all initialized local variables get executed, walking up the stack until the unwind is caught. Conveniently, Miri does not have to implement most of the logic for this: the MIR code already contains explicit instructions for which

[8]Discussing all the details of MIR here would go too far; the curious reader can find them online at https://rustc-dev-guide.rust-lang.org/mir/index.html.

$$
\begin{aligned}
\mathit{Thread} &\triangleq \{\ \mathsf{state} : \mathit{ThreadState},\\
&\qquad \mathsf{stack} : \mathit{List}(\mathit{Frame}),\\
&\qquad \mathsf{unwind\_payloads} : \mathit{List}(\mathit{Scalar}),\\
&\qquad \mathsf{data\_race} : \mathit{DataRaceThread}\ \}\\
\mathit{ThreadState} &\triangleq \mathsf{Enabled} \mid \mathsf{Blocked} \mid \mathsf{Terminated}\\
\mathit{ReturnCont} &\triangleq \mathsf{Stop}\\
&\mid \mathsf{Goto}(\mathsf{ret} : \mathit{MirBlockId},\\
&\qquad \mathsf{unwind} : \mathit{UnwindAct})
\end{aligned}
\qquad
\begin{aligned}
\mathit{Frame} &\triangleq \{\ \mathsf{body} : \mathit{MirBody},\\
&\qquad \mathsf{locals} : \mathit{MirLocal} \xrightarrow{\text{fin}} \mathit{Pointer},\\
&\qquad \mathsf{loc} : \mathit{MirBlockId} \times \mathbb{N},\\
&\qquad \mathsf{return\_place} : \mathit{Pointer},\\
&\qquad \mathsf{return\_cont} : \mathit{ReturnCont},\\
&\qquad \mathsf{catch\_unwind} : \mathit{CatchUnwind}_\perp\ \}\\
\mathit{UnwindAct} &\triangleq \mathsf{Cleanup}(\mathit{MirBlockId}) \mid \mathsf{Continue}\\
&\mid \mathsf{Unreachable} \mid \mathsf{Terminate}
\end{aligned}
$$

Fig. 3. The representation of threads and their stacks in Miri

destructors to call when. Each function call designates which block to jump to when the function returns the regular way, *and* it designates what exactly to do when the function unwinds—this is the *UnwindAct*ion, and it is stored in the return_cont of the callee so that we know what to do when that callee unwinds. Unwinding typically either has to perform some Cleanup (by executing the MIR code in the given basic block), or it can immediately Continue in the caller (*i.e.,* the current function has no destructors that must be executed). There are also dedicated actions for representing the fact that unwinding is impossible (Unreachable makes it Undefined Behavior for the function to ever unwind, which is useful information for an optimizing compiler), and that the program should be abruptly halted upon unwinding (Terminate).

Unwinding can be stopped by using the Rust standard library function `catch_unwind`, which acts much like the try-catch construct of a language with exceptions:

```rust
let res = catch_unwind(|| if valid { answer } else { panic!() });
```

Afterwards, the code can inspect `res` to check whether a panic was caught.

For Miri to implement this behavior, we leave a special marker on the stack frame of the closure invoked by `catch_unwind`: this is the catch_unwind field of Frame. For brevity's sake we do not go into detail about the exact data being stored there; what is relevant is that it lets Miri propagate the information of whether the function returned regularly or whether unwinding occurred. The last piece of the unwinding puzzle is *unwind_payloads* in *Thread*: this corresponds to the actual exception data being thrown and caught. When unwinding starts, a payload representing the current panic is appended to the list of payloads; when unwinding is caught, the last payload is removed from that list and made available in `res`. Note that we need a "stack" of payloads since more than one unwind can go on at the same time: within one of the destructors that is executed during an unwind, there may be a call to `catch_unwind` and so there could be a second panic occurring during the cleanup of the first panic. (Panicking again during a cleanup *without* such a surrounding `catch_unwind` is called a double-panic and aborts the program; this is implemented via the Terminate variant of *UnwindAct* so Miri gets this behavior "for free" by just executing MIR.)

Concurrency. And finally, we have to consider concurrency. Miri itself is a sequential program, but we can interpret concurrent programs by tracking a separate stack for each thread, and all the associated per-thread state. Execution randomly switches between threads to simulate an interleaving concurrent execution. The state field tracks whether the thread is currently ready to take another step, blocked, or has terminated. A thread can be blocked *e.g.,* when it waits for a pthread mutex that is held by another thread (see §5.2).

3.4 Soundness, Completeness, and de-facto Undefined Behavior

Having defined the key ingredients for a model for executing Rust programs, there remains a key question: is our model actually correct? Ideally, we would like Miri to be *sound* (all behaviors produced by Miri can really occur, and when Miri reports UB the program really has UB) and *complete* (all really observable behaviors can be found by Miri, and when a given execution has UB, Miri will report an error).

Generally, our strategy to ensure soundness includes comprehensive testing and discussions with compiler developers. If there was a major discrepancy between Miri and regular program execution, the large-scale evaluation in §6.1 would have likely uncovered that. We also benefit from the fact that Miri is used in practice by unsafe code authors, who will let us know when they see Miri produce outcomes that they consider impossible. (We do receive such reports regularly, but it almost always turns out to be a misunderstanding on the side of the programmer, not a bug in Miri.) Prior work on fuzzing the Rust compiler [43] compared the behavior of Miri with that of the compiled program, and while they identified plenty of compiler bugs, they did not find a single bug in Miri. In the rare case that a discrepancy between Miri and the compiler is discovered, we engage in a discussion with the Rust compiler and language teams to determine the intended behavior, and then adjust either Miri or the compiler to fix the discrepancy.

The question of completeness gets complicated by the fact that some aspects of Rust, such as the schedule chosen for a concurrent program or the exact absolute address of memory allocations, are inherently non-deterministic. Miri can only explore a single execution at a time. As discussed in the context of Example 3, Miri picks that execution based on a pseudo-random number generator. This means any given run of Miri can miss Undefined Behavior in a program if the chosen execution happens to not hit the conditions that lead to UB. While Miri supports exploring multiple independent random executions in parallel, there is no support for exhaustively exploring *all* behaviors—which, in the case of allocation non-determinism, would anyway be impractical due to the sheer size of a 64-bit address space. Furthermore, when it comes to weak memory behaviors as in Example 4, there can be outcomes that are legal in Rust but which Miri is not able to explore; we will get back to this in §4. For all these reasons, Miri can only promise completeness for *deterministic* Rust programs. For non-deterministic programs, we try to pick random exploration strategies that are more likely to hit corner cases programmers are likely to get wrong, but we cannot exclude the possibility of Undefined Behavior lurking in an unexplored execution.

A further complication arises because Rust does not yet have a full specification for what is and is not Undefined Behavior. To close the gaps in Rust's specification, we model the current *de-facto* Undefined Behavior: by inspecting how relevant language constructs get compiled, and through extensive discussions with the Rust compiler team, we can capture all the assumptions about the program that current versions of the compiler make. These assumptions may change in the future, in which case we will have to adjust Miri, but for today's compilers, to the best of our knowledge, Miri models all UB that can affect a program's correctness.

4 Finding Concurrency Bugs: Data Race Detection and Weak Memory Exploration

The state we have described so far suffices to detect almost all of the Undefined Behavior mentioned in §2, with one exception: data races (Example 3).[9] We also have not yet explained how Miri explores weak-memory behaviors (Example 4). Both of these aspects of Rust semantics are defined by the concurrency memory model, which Rust inherits from C++. Miri implements that memory

[9]Note that the Stacked/Tree Borrows aliasing check does *not* substitute a data race detector. The aliasing model deliberately does not restrict raw pointers and interior mutable shared references, so we still need a separate data race detector.

$$
\begin{aligned}
\mathit{VTimestamp} &\triangleq \mathbb{N} \times \mathit{Span} \\
\mathit{VClock} &\triangleq \mathit{TID} \xrightarrow{\text{fin}} \mathit{VTimestamp} \\
\mathit{DataRaceLoc} &\triangleq \{\, \text{read} : \mathit{VClock}, \\
&\qquad \text{write} : \mathit{TID} \times \mathit{VTimestamp}, \\
&\qquad \text{atomic} : \mathit{DataRaceALoc}_\bot \,\} \\
\mathit{DataRaceThread} &\triangleq \{\, \text{clock} : \mathit{VClock}, \\
&\qquad \text{fence\_acquire} : \mathit{VClock}, \\
&\qquad \text{fence\_release} : \mathit{VClock}, \\
&\qquad \text{write\_sc} : \mathit{VClock}, \\
&\qquad \text{read\_sc} : \mathit{VClock} \,\} \\
\mathit{DataRaceGlobal} &\triangleq \{\, \text{last\_sc\_fence} : \mathit{VClock}, \\
&\qquad \text{last\_sc\_write} : \mathit{VClock} \,\}
\end{aligned}
\qquad
\begin{aligned}
\mathit{DataRaceALoc} &\triangleq \{\, \text{atomic\_read} : \mathit{VClock}, \\
&\qquad \text{atomic\_write} : \mathit{VClock}, \\
&\qquad \text{sync} : \mathit{VClock}, \\
&\qquad \text{size} : \mathbb{N}_\bot, \\
&\qquad \text{store\_buffer} : \mathit{List}(\mathit{StoreElem}) \,\} \\
\mathit{StoreElem} &\triangleq \{\, \text{store} : \mathit{TID} \times \mathit{VTimestamp}, \\
&\qquad \text{sync} : \mathit{VClock}, \\
&\qquad \text{is\_sc} : \mathbb{B}, \\
&\qquad \text{load\_ts} : \mathit{TID} \xrightarrow{\text{fin}} \mathit{VTimestamp}, \\
&\qquad \text{sc\_loaded} : \mathbb{B}, \\
&\qquad \text{val} : \mathit{Scalar}, \,\}
\end{aligned}
$$

Fig. 4. Data race detection state in Miri

model by incorporating the C++ data race detector by Lidbury and Donaldson [26], which is based on FastTrack [8].

Unfortunately, we cannot use the data race detector described in the paper unchanged: first of all, the paper assumes a static partition of memory into atomic and non-atomic locations, which does not reflect how atomics work in modern C++ and in Rust (§4.2). Secondly, the paper was developed against the original C++11 memory model, but C++17 and C++20 both changed that model and in particular C++20 *disallowed* some behaviors that were previously permitted, so we have to adjust the dynamic data race detector to no longer generate these behaviors (§4.3).

4.1 Integrating Dynamic Data Race Detection into Miri

The Miri data race detector state can be found in Figure 4. Remember that *DataRaceLoc* is the per-byte data-race state stored inside *DataAlloc* and *DataRaceThread* is the per-thread state. We also have *DataRaceGlobal* for the global state that exists only once for the entire program.

The core data structure of the data race detector is a *vector clock*, which stores a per-thread timestamp for each thread (identified by its *TID*, the thread ID). In a weak memory system, it is not possible to correctly describe time with a single, totally ordered number. Events can be truly concurrent without being ordered either way, which vector clocks represent by using a pointwise order—crucially, this order is *not* total. Each timestamp in a vector clock is represented by a monotonically increasing number, plus a *Span* representing the program point corresponding to that timestamp. This is what allows Miri to print the user-friendly error messages demonstrated in §2, which point at *both* of the conflicting accesses that caused a data race.

Every thread has a "current clock" clock, which stores the current time from this thread's perspective. For the thread itself, that vector clock stores the latest timestamp; for the other threads, it stores the timestamp of the most recent event in that thread that the current thread has observed. Furthermore, for every location, we store which thread has most recently written to this location and when (*DataRaceLoc*.write), and we store a clock that maps each thread to the most recent time that this thread has read this location (*DataRaceLoc*.read). In FastTrack, this can be found in Figure 5 [8] in form of the `W`, `R`, and `Rvc` fields of `VarState`.[10]

With just this part of the extended interpreter state, we can detect basic data races as follows:

- On each read, we check which thread has most recently written to this location. We look up that thread in our current clock to ensure that this write occurred in our past. If it did not, this is a data race. We also update the read clock, setting the clock's value for the current thread to the latest timestamp.
- On each write, we also check the most recent write as above. Furthermore, we check that the entire read vector clock is in our past, thus ensuring that all reads have been properly synchronized. If either condition is violated, this is a data race. Finally, we update the write clock to our current clock and reset the read clock to zero.

This explains the basic data race check for non-atomic accesses. To also model atomic accesses and synchronization primitives like locks, we need a way to represent *synchronization*, which occurs each time a happens-before relationship is established between two threads. Let us consider lock-based synchronization as an example: When thread A releases the lock, its current vector clock is stored with the lock. When thread B later acquires that lock, it merges the lock's clock into its current clock (by applying a pointwise maximum). This means that all events that happened "in the past" of the lock release in thread A are now also "in the past" of thread B, thus ensuring correct behavior of the core data race check described above.

For the full algorithm, we refer the reader to the original papers [26, 8]. Here is an overview how the fields in Figures 6 and 10 [26] map to our state in Figure 4:

- Their $\mathbb{S}_{\{F,W\}}$ fields correspond to our last_sc_fence and last_sc_write.
- Their *ThrState* corresponds to our *DataRaceThread*. Their field $\mathbb{C}$ matches clock, and $\$_{\{W,R\}}$ correspond to write_sc and read_sc. Their $\$_F$ does not have an equivalent in our model due to a change in how SC fences behave (see below). Our fence_release and fence_aquire fields correspond to their $\mathbb{F}^{rel}$ and $\mathbb{F}^{acq}$ maps described in Figure 6.
- Their *StoreElem* (which will become relevant in §4.3) corresponds to ours, with their t and c field being our store, their *sc* being our is_sc, and their *clock* being our sync.
- Their *ALocInfo* corresponds to our *DataRaceALoc*, and their $\mathbb{L}$ is our sync. The remaining details differ a lot due to the update for C++20; in particular, their $\mathbb{V}$ disappears entirely.

We continue by describing the key difference between the algorithm as described by prior work, and what is implemented in Miri.

4.2 Mixed Atomic and Non-Atomic Accesses

In C++11, each memory location was statically denoted as being either atomic or non-atomic, and that fundamental distinction is baked into the C++11 data race detector. However, this distinction got relaxed in later versions of C++ with the introduction of `atomic_ref`, and Rust similarly allows mixing atomic and non-atomic accesses to the same location under some conditions. Therefore, the concept of "atomic locations" has become more subtle. In Miri, we use this term to refer to locations that are *currently* being used atomically: all locations start out non-atomic; they become atomic when an atomic access (read or write) is performed on them; and they become non-atomic again when a non-atomic *write* is performed on them. A non-atomic *read* can be freely mixed with atomic reads in Rust, and thus does not affect the location's atomicity.

[10]FastTrack's `R` field is used for an optimized representation for the read clock when there is only a single reader, avoiding the allocation of an array in `Rvc`. Since Miri is written in Rust, we can abstract this away by using a `SmallVec`, which transparently applies a similar optimization to *all* our vector clocks, in a more efficient way than would be possible in Java.

For atomic locations, Miri initializes the atomic field of *DataRaceLoc* with an instance of *DataRaceALoc* (for "atomic location"). This field tracks, for each thread, when that thread performed its most recent atomic read and write on this location. The sync clock is used for release-acquire synchronization: on a release write, this gets reset to the writing thread's current clock; and on an acquire read, it gets merged into the reading thread's current clock.

The last field, size, is used to detect *mixed-size atomic accesses*. The C++ and Rust memory models do not allow one thread to perform a 2-byte atomic access to some memory location while, at the same time (*i.e.,* without synchronization), another thread performs a 4-byte atomic access to the same location. Rust adds an exception to this rule by saying that as long as all concurrent accesses are *reads*, this is permitted. We do not have to worry about partially overlapping accesses (such as two 4-byte accesses that overlap on two bytes "in the middle") because atomic accesses always have a power-of-2 size and must be aligned to their size, *i.e.,* a 4-byte atomic access can only occur on an address that is divisible by 4. Therefore, any two 4-byte atomic accesses are either disjoint or access exactly the same memory.

With all that out of the way, here is how Miri detects unsynchronized mixed-size atomic accesses (which is, to our knowledge, a novel approach): On the first atomic access to some location, the size field is initialized to the size of this access. Note that for multi-byte accesses, there is a separate *DataRaceLoc* and with it a *DataRaceALoc* for each byte! Each byte now "knows" the size of the atomic access it has been part of. Since atomic accesses must be aligned, this uniquely identifies the exact memory range affected by the access (we can simply round the address of this byte down to the previous multiple of size to compute the first byte of the access). Later, if an atomic access happens-after *all* prior atomic accesses (which can be checked using the read and write clocks), we also freely reset the size, since there is no restriction for different sizes when there are no concurrent (unsynchronized) accesses. The interesting case is what happens for later non-synchronized accesses: an atomic read access checks whether the size of the new access matches the previously recorded size; if not, the size is reset to $\perp$. This indicates that the location may still be read by arbitrary atomic accesses, but no more writes are possible. We also check whether the current thread's clock is after the write clock; if not, then there was an atomic write to this location that conflicts with this differently-sized read, so we report UB. Finally, for atomic writes, if the size field is $\perp$ or does not match the size of the current access, we also report UB.

4.3 C++20 Weak Memory Exploration

Like the original paper [26], Miri can explore executions that involve weak memory effects by maintaining a per-location store buffer. The structure of the store buffer is shown in Figure 4.

The core principle of the store buffer is that every time we perform an atomic store, we push a *StoreElem* into this location's store_buffer. Every time we perform an atomic load, we search backwards through the location's store_buffer to collect values the ongoing load is permitted to observe. If there are multiple permitted values, we pick one randomly. Miri's store buffer differs from the original paper in the following ways:

- Their store buffer elements have a field loads that tracks all atomic loads which read-from this element. This is used to maintain read-read coherence: we need to know if the ongoing load happens after some other load. However, their implementation uses a more optimal representation, and we follow the same approach: instead of tracking all loads, we only need a field load_ts that records the earliest load per thread. This suffices because the ongoing load happens after *any* load in a thread iff it happens after the first load in that thread.
- Their *StoreElem* has a field clock that is not actually used by any of their rules. It turns out this is a mistake in the paper: this field should be used as part of the [LOAD] operation to

acquire the right clock when loading from this store buffer element. Their implementation does use this field the same way we do.

- Our implementation has been modified to account for C++20 memory model changes: we added a new field sc_loaded to *StoreElem* to account for the introduction of coherence-ordered-before, we replaced SC fences, and we adjusted relaxed stores to no longer extend release sequences.

We now go into more detail about the differences mentioned in the last bullet point.

coherence-ordered-before. The core change in C++20 is the introduction of the *coherence-ordered-before* relation (also called "extended coherence order"). The details of this are complicated and go beyond the scope of this paper [21], but the consequence of this relation and its consistency requirements can be seen in the following program (*sc* denotes sequentially consistent accesses, *rlx* is for "relaxed", and *na* denotes non-atomic accesses):

$$x :=_{na} 0;\ y :=_{na} 0$$
$$\begin{array}{r||l||l||l} x :=_{rlx} 1 & \text{spin until } 1 = x_{sc} & \text{spin until } 1 = y_{sc} & y :=_{rlx} 1 \\ & c := y_{sc} & d := x_{sc} & \end{array}$$

Under the C++11 memory model, both c and d can take the value 0 in the same execution, and the C++11 data race detector produced this behavior. In C++20, this execution outcome is forbidden. In Figure 5, relevant coherence-ordered-before edges are marked with dashed lines, and program-order edges are marked with solid lines. As per the definition of coherence-ordered-before, the dashed lines include the reads-from relation as well as the reads-before relation: if an atomic variable has been written twice, and a load sees the older value, then we say the load reads-before the newer store. The C++20 consistency predicate implies that coherence-ordered-before union happens-before is acyclic, but the combination of reads-before, reads-from, and program-order (which is included in happens-before) in this graph is cyclic, thus ruling out this execution.

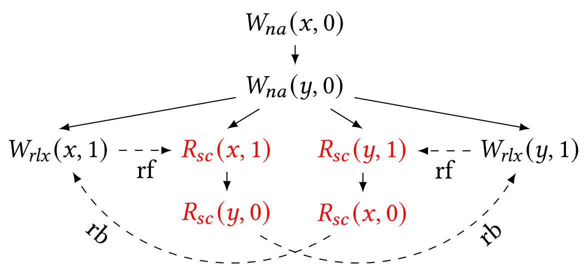

Fig. 5. Execution forbidden under C++20

To prevent this execution, we introduce an additional field sc_loaded to *StoreElem*. This is set to true whenever an SC load reads from the *StoreElem*. When assembling candidate store buffer elements for an SC read, as we search backwards through *StoreBuffer*, when we reach a value where sc_loaded is set, we stop searching further back. In the example, $R_{sc}(x,1)$ would set this flag on the store buffer element corresponding to $W_{rlx}(x,1)$, and similar for y. As a result, at least one of the following SC reads would see such a flag on the latest element in the store buffer, so they cannot both read 0. In general, this prevents inconsistent coherence-ordered-before edges between two SC loads from forming through the union of reads-before and reads-from.

SC fences. C++20 strengthened SC fences to rule out some C++11 behaviors [21]. SC fences are now equivalent to fence(Acq); rmw(f, AcqRel); fence(Rel), where f is a unique global location not exposed to the program [27]. Miri hence implements SC fences as syntactic sugar using this definition, which removes the need for the $\$_F$ field in the thread state.

Release sequences. C++20 weakened release sequences, allowing some behaviors forbidden in C++11. Relaxed stores performed by the same thread that started a release sequence no longer maintain the sequence. The original paper needed a vector of vector clocks $\mathbb{V}$ to treat relaxed stores in a thread that started a release sequence differently from stores in other threads. This is no longer needed, all relaxed stores can now be treated the same.

Limitations. This weak memory exploration inherits from its predecessor the limitation that it cannot produce *all* of the weak behaviors permitted by the C++ memory model. First of all, like all tools for verifying or testing code against the C++ memory model, Miri has to resolve the notorious out-of-thin-air issue [4] or else it would declare entirely correct programs as buggy. We follow the usual approach of ruling out cycles in the union of the program-order and reads-from relations [21]. This excludes some load buffering behaviors that are allowed by the C++ memory model.

On top of that, Miri also cannot generate executions that contain a cycle in the program-order and modification-order relations. The reason for this is that Miri fundamentally still executes all operations in a single, global order, and the modification-order of each memory location is a subset of that global order. For instance, the classical 2+2W litmus test [35] will never produce a final state where both x and y are 1, despite that outcome being allowed in C++20 (if all stores use release or weaker ordering). This means Miri misses some bugs, but it will not report false bugs.

5 Making Miri a Practical Tool

Miri as we have described it so far would be a perfectly functional Undefined Behavior detector, but it would not be a practical tool: it would be too slow, require too much memory, and it would not be able to run most real-world Rust code unchanged. The trouble with real-world code is that it will inevitably call some functions that are not implemented in Rust, but imported from system libraries typically written in C: most programs print output to the terminal, open files, use synchronization primitives, or use other common OS facilities such as environment variables and thread-local state. Another problem is created by programs using CPU-vendor-specific intrinsics, such as the SSE and AVX families of intrinsics on x86 (Intel/AMD) CPUs.

In this section, we describe how we overcome those problems. To increase the performance of Miri, we alter the architecture described in §3 to represent certain common cases more efficiently. As we will see in §6, this does not exactly make Miri *fast*, but it makes Miri *fast enough* for practical use. And to support real-world programs that make use of non-Rust system functions and intrinsics, we equip Miri with implementations for the most widely-used operations.

5.1 More Efficient Representation of Common Cases

Looking at the architectures described in §3, there are several glaring inefficiencies:

- The biggest issue is the *DataAlloc*.data field (Figure 2). This stores the contents of an allocation as a naive list of *Byte* and *DataRaceLoc*. Both of these types have a non-trivial size, so a single byte of machine memory needs many, many bytes of interpreter memory.
- Another problem is caused by *Operand*: if, for instance, a Rust program does an assignment of a 1 KiB array, then the naive architecture described so far would have us first allocate a 1 KiB array on the heap, copy the original data into that new array, and then copy that data into the destination of the assignment.
- The *VClock* type used extensively by the data race detector would be too large if it were naively implemented as a hash map.
- And finally, there is the problem that every single local variable is represented by a *Pointer* to memory. The MIR generated by the Rust compiler contains many temporary variables that do not show up in the source program. This is common for low-level IRs generated from high-level programs, and typically most of those variables disappear during optimizations, but Miri turns off all optimizations to avoid missing any Undefined Behavior.

$$
\begin{aligned}
\mathit{DataAlloc} \triangleq \{\ & \mathsf{bytes} : \mathit{List}(\mathbb{N}_{<256}), \\
& \mathsf{init\_mask} : \mathit{Bitvector}, \\
& \mathsf{prov\_ptrs} : \mathit{List}(\mathbb{N} \times \mathit{Provenance}), \\
& \mathsf{prov\_bytes} : \mathit{List}(\mathbb{N} \times \mathit{Provenance}), \\
& \mathsf{data\_race} : \mathit{DedupRangeMap}(\mathit{DataRaceLoc}), \\
& \mathsf{mutable} : \mathbb{B}, \quad \mathsf{align} : \mathbb{N}\ \}
\end{aligned}
\qquad
\begin{aligned}
\mathit{Operand} &\triangleq \mathsf{Scalar}(\mathit{Type} \times \mathit{Scalar}) \\
&\mid \mathsf{Indirect}(\mathit{Type} \times \mathit{Pointer}) \\
\mathit{Frame} &\triangleq \{\ \mathsf{locals} : \mathit{MirLocal} \xrightarrow{\text{fin}} \mathit{Operand}, \\
&\qquad \ldots\ \}
\end{aligned}
$$

Fig. 6. Optimized representation of memory and values in Miri

We now discuss the architectural choices Miri makes to mitigate these problems.

More efficient representation for Byte. Looking at the *Byte* type, the first and most obvious problem is that even if we ignore provenance, a *Byte* can hold 257 distinct values, and thus cannot be stored in 8 bits of host memory. To avoid needing to waste an entire byte to store a single bit of information, we split the three components of *Byte* into three separate lists: an array of `u8` just for the raw data, a bitvector (making full use of the 8 bits in each byte) for representing which parts of the data are initialized, and a list for the provenance. Furthermore, we make the provenance list *sparse*, *i.e.*, it is represented as an ordered list of offset-provenance pairs. This means that for memory that does not contain any pointers, we have a memory overhead of only 1/8th (the initialization bit for each byte), which is optimal for non-compressed representations.

However, the provenance representation can be further optimized: almost all the time, provenance occurs in a pointer-sized chunk of adjacent bytes carrying the same provenance. Actual per-byte provenance is only needed if the program starts copying around raw bytes one at a time, which is very rare. This was already visible in Example 7 at the end of §3.2. Therefore, instead of a single sparse provenance map, we have two: one for pointer-sized provenance, and one for byte-sized provenance. The final representation of memory can be found in Figure 6.

More efficient representation for DataRaceLoc. The inefficiencies in *DataRaceLoc* arise from the fact that most of the time, memory is accessed in chunks of more than one byte. For instance, if the program has a `struct` with a lot of fields of type `i32`, all accesses to those fields will happen 4 bytes at a time. This means that the data race metadata for the 4 bytes that correspond to a single field will always be the same. To exploit this redundancy, we store *DataRaceLoc* in a *DedupRangeMap*. This data structure conceptually works like an array, but the main API for mutating the array is not indexing, it is iterating over a *range* of the array and mutating that range in-place. Under the hood, the map stores contiguous ranges of the same value in a compact form, and dynamically splits those ranges when the edges of such an updating iteration fall into the middle of a previously contiguous range of values. This ensures that if some chunk of memory is only ever accessed in 4-byte reads and writes, it will never be split, saving both space and time (since there are fewer vector clocks to check on each access).

More efficient non-scalar Operand. To avoid the intermediate buffer and the associated double-copying that comes with a large *Operand*, we introduce the idea of *indirect* operands: essentially, we delay actually loading the memory from the source of an assignment until we have figured out where the result should be stored, and then we directly copy the bytes from source to destination. Obviously, this means we have to be very careful to not *change* the contents of memory that the indirect operand points to; the operand is conceptually a separate value so such a change would break the semantics. This is where we benefit from MIR having a very simple and regular structure:

there are no nested expressions and evaluating an expression to an operand can never change the contents of memory. The optimized version of *Operand* can be found in Figure 6.

More compact VClock. The "obvious" choices for representing *VClock* would be a hash map from thread IDs to timestamps, or a vector of timestamps. To avoid the overhead of a hash map, we are using a vector. However, a naive vector would bloat over time as more and more threads get spawned and the threads represented by low thread IDs terminate and stop mattering. To deal with this, we maintain a dynamic mapping from thread IDs to *vector clock IDs*: when a thread finishes, it donates its vector clock ID into a pool, making it available for reuse. If later, a thread spawns in a way that fully happens-after the end of the previous thread, it can just reuse the same vector clock ID. For code that keeps spawning and joining threads, this means the vector clock IDs remain small, and therefore vector clocks remain compact.

More efficient representation for local variables. Finally, we turn to the problem of local variables being inefficient. To mitigate this, we introduce an optimized representation for *Scalar* variables. In particular, all integer and pointer types as well as **`bool`** and fieldless **`enum`** fit in *Scalar*, so this covers a significant fraction of variables. That said, we have to restrict the optimization to variables that do not have their address taken: once a variable has its address taken, we need to be able to construct a *Pointer* to it, so it needs to be stored in addressable memory. But before that happens, there is actually no reason for it to be stored in an *Alloc*. Instead, we change the type of the locals map in *Frame* to store, for each local variable, *either* the place in memory where this local is stored, *or* its actual current value for the optimized representation. It turns out that "either a scalar or in memory" is already exactly represented by *Operand*, so we reuse the same type for this—which makes sense, after all this map stores the current *value* of the local variable, and as discussed above *Operand* is the type of values in Miri. Variables of scalar type dynamically switch to the less optimal representation when their address is taken. When this happens, we have to take care to set the vector clocks associated with the memory holding the local variable to the last time the variable was actually read/written, not the moment the variable moved into memory.

5.2 Supporting Non-Rust Operations

Here we give an overview of the operating system and hardware vendor operations that are implemented in Miri. This represents a significant chunk of the work of building Miri: more than one third of the source code of Miri implements various non-Rust operations so that more of the Rust code people care about can be executed with Miri.

However, before we go over the main APIs supported by Miri, we need to briefly explain some general aspects of such target-specific operations. Miri is a fully cross-target interpreter. This means a user can be on a Windows computer and invoke Miri with `--target x86_64-unknown-linux-gnu`, and Miri will build and run the code as-if it was running on Linux. In this case, we would say that the *target* is Linux and the *host* is Windows. Miri capabilities generally are fairly independent of the host: almost all of Miri is implemented on top of the Rust standard library, and as such works on any host OS that Rust supports. This ability to run code for other targets is particularly useful for testing more exotic configurations, such as targets with 32-bit pointers or big-endian targets. But this also means that to support an operation like writing to a file across all targets, we need to implement all the target-specific APIs for this! Generally, the best-supported target in Miri is Linux, and more broadly we support all POSIX targets reasonably well since their APIs heavily overlap with Linux. We actively test this on macOS, FreeBSD, and Illumos/Solaris. Windows is supported, too, but lacks some of the operations whose equivalent Linux functions are supported.

Files and file descriptor/handle manipulation. Miri supports the basic operations to open, read, write, and close files. To achieve this, Miri has its own internal table of file descriptors (or handles, as they are called on Windows), and various kinds of file descriptors that implement the common interface required by the table. In particular, this also includes the standard input, output, and error streams, which is crucial for pretty much any program.

Beyond basic files, we support a few more kinds of file descriptors: On all POSIX targets, we support pipes and anonymous sockets (but only streaming sockets, not datagrams). On Linux, we also support `epoll` and `eventfd`. This is important since it lets us execute the Tokio runtime: Tokio is the most widely used runtime for asynchronous Rust code, so a significant fraction of the ecosystem needs Tokio support for testing anything at all. However, Miri does not currently support network sockets, so only the core processing loop of asynchronous Rust programs can be tested at the moment, not the parts that communicate with the outside world.

File system manipulation. On POSIX systems, Miri also supports listing directories and renaming and removing files. Directory listing is a stateful API, so this requires some bookkeeping in Miri to properly map the `readdir` call we are emulating to the Rust standard library type for iterating over directory contents.

Concurrency APIs. Miri supports the basic OS functions for spawning threads; this is crucial for testing any concurrent program. We also support the pthread synchronization primitives. However, those are actually not used by Rust any more (on most targets), so on top of this we support various OS-specific APIs: the futex syscalls on Linux, the "unfair lock" on macOS, as well as the Windows and FreeBSD variants of futexes (`WaitOnAddress` and `UMTX_OP_WAKE`, respectively).

All of these APIs need to interact properly with the data race detector. And, in fact, `epoll`, pipes, and anonymous sockets can also be used to communicate between different Rust threads, so those must also be recognized by the data race detector. We do this with a general API that allows the implementation of OS-specific operations to obtain a vector clock by calling "release", and to later "acquire" that vector clock on a different thread when synchronization has occurred. This avoids polluting the data race detector with the details of all these APIs.

Thread-local state. Miri supports both the POSIX and Windows APIs for allocating and managing thread-local state (TLS). This is needed to support the widely-used `thread_local!` macro from the standard library. The main difficulty here is dealing with TLS destructors: all platforms support some form of registering code to be run when a thread exits, so that the memory allocated for thread-local state can be freed again. Therefore, when the main function of a thread returns and the thread's stack is empty, we are not done yet: we need to figure out if there are thread-local destructors to run, and invoke them one after the other, before fully considering this thread to be done. This may have to happen in a loop; POSIX TLS destructors are allowed to re-initialize other TLS variables whose destructors will then be called a second time. To keep this complexity manageable and localized, we build up a state machine for "what to do the next time this thread's stack is empty". The main interpreter loop only has to drive this state machine, while the implementation of the state machine can be entirely confined within the module that deals with thread-local state.

Timekeeping. Miri supports the usual APIs for getting the current system time, and for putting a thread to sleep for a given amount of time. This is less trivial than it may sound since it impacts the scheduler: if a thread goes to sleep, we have to keep scheduling other threads—until *all* threads are sleeping; at that point we have to actually sleep Miri itself until the first thread wakes up. We also support APIs with a timeout (such as waiting on a futex with a timeout), including APIs that let the user select different sources of time: a timeout can be tied to the system clock, or to some unalterable strictly monotone clock that always counts up even if the system time gets changed.

This is all implemented with a general mechanism of "blocking a thread with a timeout", and an associated "unblock callback" that is invoked by the scheduler either when the timeout fires or when the thread gets unblocked by other means. This keeps the logic for all the individual operations separate from each other and from the scheduler, while also keeping the code that marks a thread as blocked very close to the code that runs after unblocking (the callback is always defined in-line as a closure), which greatly simplifies reasoning about this code.

Intel vendor intrinsics. The `std::arch` module of the Rust standard library provides Rust programs with direct access to many architecture-specific operations. Since Rust does not yet have a standard API for portable SIMD operations, quite a few libraries use `std::arch` to accelerate their core loops with manual use of SSE/AVX operations. To support such programs, Miri has support for SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, and AVX2, plus partial support for the cryptographic intrinsics in the AES-NI, SHA, and GFNI extensions. Lucky enough, Miri does not have to directly implement *all* the many hundreds of intrinsics in these extensions; many of them are implemented in `std::arch` by means of a few general SIMD operations that Miri can support once and for all. Still, Miri has more than 100 separate implementations for Intel-specific intrinsics.

. . . and a long tail. There is a long tail of operations that are needed to cover even just the core parts of the Rust standard library, sometimes for surprising reasons. For instance, some standard library operations check whether a particular environment variable is set, so Miri has a full implementation of the process environment. The standard library `HashMap` uses a random seed to mitigate HashDoS attacks, so Miri needs to support various APIs for obtaining strong random numbers from the OS. Some operations the standard library wishes to use are not supported on all versions of the target OS, so they use `dlopen`/`GetProcAddress` and a mechanism called "weak symbols" to dynamically make use of these functions only if they are present—even more APIs for Miri to support. And finally, we also support many of the standard C functions, such as `malloc`, `free`, `exit`, since they are often used to implement the corresponding Rust functionality.

6 Evaluation

We evaluate the effectiveness of Miri in three different ways: we perform a large-scale experiment on the Rust ecosystem to determine which fraction of the Rust code out there Miri is able to check for bugs, we run Miri on a microbenchmark to quantify the overhead of Miri compared to regular execution, and we report on some of the real-world bugs that Miri has found.

6.1 Compatibility with the Rust Ecosystem

To systematically evaluate the compatibility of Miri with the Rust ecosystem, we downloaded the 100 778 crates with the highest number of recent downloads from crates.io (the central registry of the Rust ecosystem), and ran their test suites in Miri. The test run covers more than half of the crates.io registry (which totals around 180k crates), but for lack of time we were not able to test every single crate. Rust crates can have three kinds of tests: unit and integration tests are functions marked with `#[test]` that are either embedded in the crate (with access to private fields), or separate code accessing the crate's public API. Documentation tests are examples integrated in the documentation, which also get tested.

We used the version of Miri distributed with `rustc 1.90.0-nightly (35f603652 2025-06-29)`. For the purpose of this test, we disabled Miri's memory leak checker. We also turned on layout randomization, which is a feature of the Rust compiler that will permute the layout of data types to uncover unsafe code making incorrect assumptions about type layout. We run unit and integration tests with cargo-nextest [33], with a per-test timeout of 60 seconds. The test run for the entire crate is subject to a 1-hour timeout and a maximal memory consumption of 8 GiB. The documentation

Outcome	Number of crates
Success	30 651 (51.8%)
Undefined Behavior	4595 (7.8%)
Out of memory	5
Miri crash	4
Unsupported	12 853 (21.7%)
Doctest build failure	4629 (7.8%)
Timeout	3478 (5.9%)
Other failure	2921 (4.9%)
Total (after filtering)	59 136

(a) Per-crate results

Outcome	Number of tests
Passed	1 103 741 (70.3%)
Failed	353 264 (22.5%)
Timed out	112 480 (7.2%)
Total	1 569 485

(b) Per-test results

Fig. 7. Results of crates.io evaluation

```
let v = (0..n).into_iter().collect::<Vec<_>>();
for i in v { *i += 1; }
```

Fig. 8. The benchmark program

tests are run separately, with just the 1-hour timeout. All of this gets executed on an AMD 3970X (32 cores / 64 threads). The run took 11 days, which amounts to approximately 1.9 CPU-years. We have filtered out crates that did not build, or that had no tests.

The results of this run can be found in Figure 7a. After filtering, there were 59 136 crates left. Each crate is put into exactly one category, with the order of categories in the table reflecting the priority: if a crate causes Undefined Behavior in one test and another test times out, it is categorized as Undefined Behavior. The main result is that about half of the crates are successfully checked by Miri. Around 20% of crates still run into unsupported non-Rust operations. While we implement most of the widely-used basic APIs, there is an extremely long tail of C libraries Rust programs use that we will never have implementations for (*e.g.*, OpenSSL, SQLite, CPython). The few Miri crashes are bugs in Miri that we have not yet managed to track down. "Doctest build failure" indicates that the crate's documentation tests did not build; we left those crates in the evaluation since all the regular tests passed Miri (or timed out). "Other" failures include tests that checked an incorrect assertion; this might be a flaky test, or it might be a sign of Miri hitting an execution path that regular testing did not cover (*e.g.*, via weak memory exploration).

We have also separately categorized the individual *tests*, collected across all crates, in Figure 7b. Overall, less than 23% of the tests fail in some way (*e.g.*, due to UB, an unsupported operation, or an assertion failure), and less than 8% of the tests time out. More than 70% of the tests pass.

6.2 Performance

For this evaluation, we run a simple benchmark on various sizes and compare the runtime when executing the program with Miri with an execution of the program compiled to native code. The program can be found in Figure 8; all it does is allocate a large array filled with the numbers from 0 to `n`, and then iterate over that array to do some arithmetic on each element.

Miri and the Rust compiler are both taken from the `rustc 1.90.0-nightly (11ad40bb8 2025-06-28)` Rust toolchain. Miri was executed with default settings, which check for all UB (including Stacked Borrows). For the native build, we disabled all optimizations (`-Zmir-opt-level=0 -Copt-level=0`). The benchmark was executed with n values ranging from 50k to 400k in steps

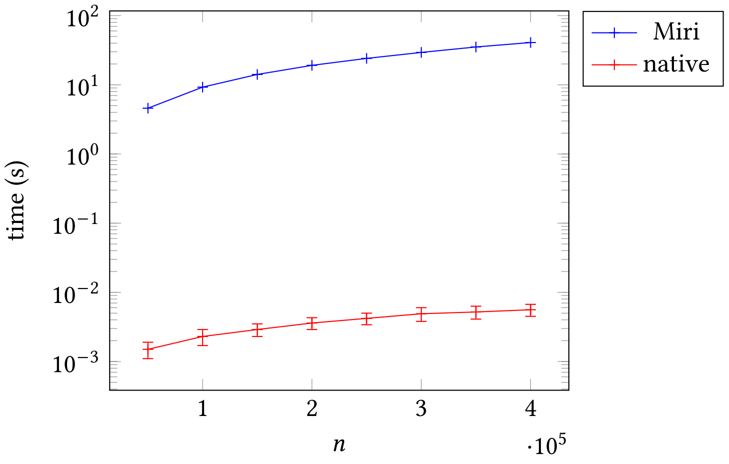

Fig. 9. Benchmark results. Error bars indicate standard deviation; we omit them for the Miri where the error is tiny compared to the runtime.

of 50k. All benchmarks were executed with hyperfine [6], gathering the average and standard deviation. Miri benchmarks were executed 3 times; native benchmarks were executed at least 100 times (hyperfine automatically runs the program more often if the benchmark is very fast, aiming for around 3s of total benchmarking time). Everything was executed on an Intel Core i7-12700H.

The results can be found in Figure 9; note that the y-axis uses a log scale. For the lowest n, Miri is about 3000x slower than the native code; for the highest n, the ratio is about 7000x. This might seem excessively slow, but computers are fast, so Miri can still be used for real-world test suites, at least after reducing iteration counts for long-running tests (as also demonstrated by this being common practice). Unfortunately, however, the overhead does make Miri too slow for typical fuzzing setups.

6.3 Real-world Bugs

In this section, we discuss some of the real-world bugs that Miri has found. All of them have been fixed in the latest versions of the affected libraries. This is not a systematic analysis; it should be taken as anecdotal evidence that Miri is able to find real bugs that maintainers care about. It is worth noting that many of these bugs have not been found by us; they have been found by other people using Miri either on their own codebase or on libraries that they care about. This is possible because Miri is distributed as part of the official “nightly” Rust toolchain. At this point, many widely-used crates and even the Rust standard library run Miri as part of their continuous integration, which means any bugs that Miri flags are prevented from ever entering the codebase. As a result, our list is by no means exhaustive: we know of plenty more bugs Miri has found (some are listed in the Miri README contained in the artifact [16]), and there are surely numerous bugs Miri has prevented without us learning about it.

Figure 10 provides an overview of the bugs we will discuss. For lack of space, we can only briefly explain what was wrong in each case, and which feature of Miri was crucial in finding the bug.

Bug 1. A bug in the debug printing logic for the iterator of the standard library `VecDeque` meant that it tried to print the contents of uninitialized memory. Miri found this via the initialization tracking that also triggered in Example 1.

#	Github issue/commit	Summary
1	rust-lang/rust#53566	accessing uninitialized memory
2	rust-random/rand#779	unaligned reads
3	rust-lang/rust#62251	calling `posix_memalign` in an invalid way
4	tikv/tikv#7751	out-of-bounds pointer arithmetic
5	rkyv/rkyv@a941719	constructing a `Box<[u8]>` from an overaligned allocation
6	matklad/once_cell#186	incorrect use of `compare_exchange_weak`
7	rust-lang/rust#124281	Weak-memory-induced memory leak
8	rust-lang/rust#141248	Weak memory ABA bug due to address reuse

Fig. 10. A selection of real-world bugs found by Miri

Bug 2. The standard random number crate used across the Rust ecosystem performed an unaligned memory access. Interestingly, the programmers seemed to have been aware that alignment is a problem in this case: there were dedicated code paths for x86 and for other architectures. Other architectures used `read_unaligned`, but the x86 code path had a comment saying that x86 allows unaligned reads, so we do not need to use this (potentially slower) operation. Unfortunately, this is a misconception: even though x86 allows unaligned accesses, Rust does not, no matter the target architecture—and this can be relevant for optimizations. Miri checks proper alignment on every memory access, thus quickly finding this bug.

Bug 3. The Rust standard library allocates memory using `posix_memalign` if the caller requests memory with an unusually high alignment, or whenever the requested alignment is smaller than the requested size. However, `posix_memalign` requires the requested alignment to be a multiple of the pointer size. When making a small but highly-aligned allocation, Rust would violate this precondition. This got found when we added support for `posix_memalign` in Miri, carefully checking all the preconditions listed in the documentation. Our test suite tries various unusual combinations of sizes and alignment, and among them was a pair that triggered this issue.

Bug 4. TiKV is a distributed transactional key-value database. The developers are testing the core parts of their algorithm using Miri, and as part of this they found a bug involving pointer arithmetic: The `add` method on pointers in Rust implements *inbounds* pointer arithmetic, meaning that it is Undefined Behavior to call this operation unless the pointer starts out within the bounds of an allocation and remains inside that same allocation. This condition was violated by TiKV because developers thought that being inbounds only becomes relevant once the pointer is actually used for a memory access. Miri checks all the requirements of `add` each time it is invoked, thus detecting the out-of-bounds pointer.

Bug 5. rkyv is a zero-copy deserialization framework for Rust, and it had a subtle bug that was found via Miri. The bug is related to Rust's `Box` type, which is the type of heap-allocated pointers. When such a pointer goes out of scope, the memory on the heap gets deallocated. The Rust allocator API is based on passing the size and alignment of the allocation alongside the to-be-freed pointer; this can help the allocator save some work when identifying where the metadata for this allocation is stored. However, any mismatch of the size or alignment causes UB. The problem in rkyv was that memory was allocated with an alignment of 16, but got converted into a `Box<[u8]>`. When the `Box` goes out of scope, it uses its pointee type to determine the alignment used for deallocation; in this case, that turns out to be 1. Therefore, the alignment on deallocation does not match the alignment on allocation, which is UB. Miri checks those alignment requirements, making it easy to find the bug once the relevant code path was executed by the test suite.

Bug 6. The once_cell crate provides a type of single-assignment thread-safe variables. It made use of `compare_exchange_weak`, a variant of `compare_exchange` that is allowed to spuriously fail even if the current value in memory actually matches the expected value. However, the logic did not correctly deal with the case where that would actually happen. Miri randomly introduces spurious failures in `compare_exchange_weak` with a fairly high probability (80% by default), and so it quickly found an execution where the program failed to properly deal with a spurious failure.

Bug 7. A subtle bug in the implementation of thread-local storage on Windows in the Rust standard library led to memory sometimes not being properly deallocated. The original code would *first* do a release store to an atomic variable `self.key`, and *then* called a function to add itself to a list of destructors for thread-local state. This means that even if another thread later observes the updated key, the thread might read an outdated value when checking which destructors are registered, thus failing to properly deallocate the associated memory. Miri's weak memory exploration caused such an outdated load, and then the memory leak check triggered.

Bug 8. This bug occurred in the implementation of a reentrant lock in the Rust standard library. For targets without 64bit atomics (such as 32bit MIPS targets), the code cannot use its usual approach to identifying which thread currently owns the lock, since the thread ID is 64bits in size. (A 32bit ID space would be exhausted too quickly.) To deal with that, the implementation uses a clever trick: each thread allocates a 1-byte piece of thread-local storage, and the *address* of that thread-local variable serves as the identifier for the current thread. The address can get reused over time when many threads are spawned and destroyed, but it must be unique among all currently running threads. However, a subtle ordering issue in the logic for checking the current owner meant that if a thread re-uses the address of a previous thread, the code only properly synchronized with the actions of the previous thread *after* already doing the first atomic operations that determine the current owner. This means those atomic operations could load outdated values, leading to a data race. Miri was able to find it because it properly implements thread-local storage, including a random chance to reuse the address of some previously freed memory. Together with Miri's weak memory exploration, that made it possible to hit this very subtle issue in Miri with a chance of about 5%; enough to identity the root cause and fix the issue.

7 Related Work

There is a wide range of tools for verifying (and not just testing) Rust code [1, 7, 13, 9, 22, 23, 10, 2]; however, those all require significant amount of annotations. Kani [17] stands out as a bounded model checker requiring very limited annotations, but it does not capture Rust's semantics to nearly the same level of precision as Miri: it does not have pointer provenance, does not support concurrency or weak memory, and conservatively rejects some raw pointer usage that does not actually cause UB. Therefore, we focus on existing tools that *test* for UB in C, C++, and Rust.

Sanitizers. As already mentioned in the introduction, the most widely-used tools for UB detection are sanitizers, in particular the family of sanitizers integrated into LLVM. These sanitizers come in various flavors: AddressSanitizer detects out-of-bounds and use-after-free issues, LeakSanitizer detects memory leaks, MemorySanitizer detects uses of uninitialized memory, ThreadSanitizer detects data races, and UBSanitizer detects issues such as signed integer overflow, out-of-bounds bit shifts, and use of misaligned or null pointers. They all work fundamentally in the same way: once the program has been compiled into LLVM IR, a transformation pass adjusts the IR to insert extra checks and bookkeeping of the metadata needed for those checks. Then compilation proceeds as normal. This makes sanitizers very fast (they all have less than 10x overhead), but also limits their ability to detect Undefined Behavior: if the frontend is doing any of its own transformations under the assumption that there is no UB (which, for instance, Rust does when it performs optimizations directly on the MIR), that can lead to UB not being detected. Even if there are no such transformations, there is almost always some information loss when translating the surface language to LLVM IR, meaning that some programs with UB can be translated to entirely well-defined LLVM IR that no sanitizer will complain about. Such programs are ticking time bombs, since a future update to the compiler can change the generated LLVM IR and thus introduce UB into programs that previously passed the sanitizer. Not all sanitizers can be turned on at the same time, so to get maximal UB checking, the program needs to be compiled and executed multiple times. Even then, not all of the relevant UB is covered: there is no sanitizer checking pointer provenance or type invariants at runtime, and bugs like bug 3 in the previous section also cannot be found by a sanitizer (that would require a hardened libc with more precondition checks). And finally, sanitizers are not able to exhibit rare non-deterministic behavior akin to Miri's weak memory implementation, putting issues such as bugs 6–8 in the previous section completely out of their reach.

Valgrind. Valgrind takes the normal binary produced by a regular build, and executes it with extra monitoring by dynamically instrumenting the machine code. Compared to sanitizers, this has the advantage of being very easy to use (no need to recompile the program with special flags), but the downsides of sanitizers are exacerbated: since the program passes through the entire usual compilation pipeline, any optimization along the way risks hiding Undefined Behavior. The final assembly code contains even less information about source-level UB than the LLVM IR. As a result, Valgrind can generally find strictly fewer bugs than sanitizers. The overhead is significantly higher, but still much lower than that of Miri—the typical slowdown is on the order of 20x–50x.

Interpreters for Undefined Behavior detection. Besides Valgrind and sanitizers, there are also some less widely-known tools specifically for detecting Undefined Behavior in C: the TrustInSoft Analyzer [40], and Runtime Verification's RV-Match [34]. Both are commercial tools, so not much is publicly documented about their architecture, performance, and capabilities.

RV-Match is based on a formalization of the C11 semantics in the K framework [12]. The K framework can turn a formal language semantics into an interpreter automatically (among other things), so this is most likely the core of RV-Match. One issue with this formalization is that it was developed entirely independent of the main compiler projects (GCC and clang). In an ideal world, this would not matter: an unambiguous specification ensures that compilers and other tools can be developed independently against the specification. However, the C standard is far from unambiguous [30, 19], and the K formalization of C11 does not discuss how they resolve all the open questions in the C standard. In contrast, Miri is developed in close collaboration with the Rust compiler team, ensuring a consistent interpretation of the semantics of surface Rust. It is unclear if Runtime Verification is even still selling RV-Match; the tool has largely disappeared from their website and the promotional video was removed from YouTube.

The TrustInSoft Analyzer is a tool described as being able to find Undefined Behavior bugs in C code. The same company used to work on a project called TIS interpreter, traces of which can still be found in the Software Heritage Archive [41]. It therefore seems likely that the core of the TrustInSoft Analyzer is what used to be the TIS Interpreter.

These tools will face challenges similar to Miri in terms of non-determinism, performance, and availability of system APIs such as file system access and concurrency primitives, but the public documentation does not suffice to make any sort of comparison here. That said, both of these tools clearly target C, so Miri is to our knowledge the only Undefined Behavior checker for Rust.

Aside from these commercial tools, several efforts of formalizing variants of the C semantics have also produced interpreters that directly reflect the formal semantics [31, 24, 20, 29]. However, none of them have been developed into practical tools for Undefined Behavior detection.

8 Conclusion and Future Work

We have presented *Miri*, a testing tool for finding Undefined Behavior bugs in Rust by dynamically checking a single program execution for UB. At its core, Miri is a direct implementation of an Abstract Machine for Rust: it is the result of taking a tool from PL theory and translating it into the real world. When combining the techniques described in this paper with Stacked Borrows or Tree Borrows, Miri is the only tool that can find *all* de-facto Undefined Behavior in deterministic Rust programs. Rust does not currently have a stable specification of what exactly is and is not UB, so we are working closely with the Rust project to ensure that Miri's idea of de-facto UB remains up-to-date. For non-deterministic programs, Miri can do random exploration to search for UB.

Since we developed Miri in the open over many years, several papers have already been published that directly make use of Miri. McCormack et al. [28] coupled Miri with an interpreter for LLVM IR to be able to run Miri on Rust programs that call C or C++ code, by means of compiling that C/C++ code with clang and interpreting the generated LLVM IR. This uncovered several bugs in widely-used libraries. SyRust [37] and Crabtree [36] stress-tests Rust libraries that contain unsafe code by automatically generating well-typed clients and executing them in Miri; these approaches were also able to find new bugs. Rustlantis [43] used Miri as an oracle to ensure that the automatically generated Rust programs (meant to fuzz the Rust compiler) do not contain Undefined Behavior. Amazon used Miri as part of their validation effort for ShardStore [5].[11]

The main weaknesses of Miri are its low performance, and the fact that Miri cannot execute Rust programs that call native libraries via FFI. Furthermore, the nature of Miri limits it to testing one execution of the program at a time; in highly non-deterministic programs, this can make it tricky to actually hit the problematic code path. In future work, we plan to further develop the ongoing experiment of having Miri call native code while still checking Undefined Behavior on the Rust side as much as possible. For drastically improving Miri's performance, it will likely be necessary to fundamentally alter its architecture: one could imagine a version of Miri built as a sanitizer, with enough compiler support to avoid losing any relevant information along the compilation process. To improve test coverage in concurrent programs, we are considering combining Miri with model checking techniques to be able to systematically explore all behaviors of a concurrent program that are permitted by the C++ memory model. And of course, there are always more platform APIs to support; in particular, support for network sockets would let Miri cover much more of the asynchronous Rust ecosystem.

Despite these weaknesses, Miri has found numerous real-world bugs and has been integrated into the official Rust "nightly" distribution and the online Rust playground. Over 70% of the tests in the 100 000 most popular Rust libraries can successfully be executed in Miri. It is part of the continuous integration of the Rust standard library and some of the most widely-used crates. In online discussions about unsafe Rust, it has become common for people to reference Miri. Overall, it is fair to say that the Rust community has adopted Miri as one of its standard tools.

Data Availability Statement

Miri is open-source software available at https://github.com/rust-lang/miri, licensed under MIT and Apache2. All ingredients required to reproduce the evaluation can be found in the artifact [16].

References

[1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573

[11] In fact, the initial support for concurrency in Miri was contributed by the ShardStore validation effort. We have significantly extended it with a random scheduler, support for detecting data races, and weak memory exploration.

[2] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2025. A Hybrid Approach to Semi-automated Rust Verification. *Proc. ACM Program. Lang.* 9, PLDI, Article 186 (June 2025), 23 pages. doi:10.1145/3729289

[3] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. 55–66. https://doi.org/10.1145/1926385.1926394

[4] Hans-Juergen Boehm and Brian Demsky. 2014. Outlawing ghosts: avoiding out-of-thin-air results. In *MSPC@PLDI*. ACM, 7:1–7:6. doi:10.1145/2618128.2618134

[5] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *SOSP*. ACM, 836–850. doi:10.1145/3477132.3483540

[6] David Peter . Last accessed July 10th, 2025. Hyperfine. https://github.com/sharkdp/hyperfine.

[7] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *ICFEM (Lecture Notes in Computer Science, Vol. 13478)*. Springer, 90–105. doi:10.1007/978-3-031-17244-1_6

[8] Cormac Flanagan and Stephen N. Freund. 2010. FastTrack: efficient and precise dynamic race detection. *Commun. ACM* 53, 11 (2010), 93–101. doi:10.1145/1839676.1839699

[9] Nima Rahimi Foroushaani and Bart Jacobs. 2022. Modular Formal Verification of Rust Programs with Unsafe Blocks. *CoRR* abs/2212.12976 (2022). doi:10.48550/ARXIV.2212.12976

[10] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. doi:10.1145/3656422

[11] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. 2022. *Programming languages – A Provenance-aware memory object model for C.* Technical Report. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3057.pdf

[12] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the undefinedness of C. In *PLDI*. 336–345. https://doi.org/10.1145/2737924.2737979

[13] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. https://doi.org/10.1145/3547647

[14] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. doi:10.1145/3371109

[15] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (April 2021), 144–152. https://cacm.acm.org/magazines/2021/4/251364-safe-systems-programming-in-rust/fulltext

[16] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Artifact for Miri: Practical Undefined Behavior Detection for Rust. doi:10.5281/zenodo.17334726 (latest version available on paper website: https://plf.inf.ethz.ch/research/popl26-miri.html).

[17] The Kani Developers. 2022. The Kani Rust Verifier. https://github.com/model-checking/kani Last accessed 01 July 2025.

[18] Paul Kehrer. 2019. Memory Unsafety in Apple's Operating Systems. https://langui.sh/2019/07/23/apple-memory-safety/.

[19] Robbert Krebbers. 2015. *The C standard formalized in Coq.* Ph. D. Dissertation. Radboud University Nijmegen. https://robbertkrebbers.nl/thesis.html

[20] Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *CPP*. ACM, 15–27. doi:10.1145/2676724.2693571

[21] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*. 618—-632. doi:10.1145/3062341.3062352

[22] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. https://doi.org/10.1145/3586037

[23] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *PACMPL* 7, PLDI (2023), 1533–1557. doi:10.1145/3591283

[24] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. 42–54. doi:10.1145/1111037.1111042

[25] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2.* Technical Report RR-7987. Inria. https://hal.inria.fr/hal-00703441

[26] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *POPL*. ACM, 443–457. doi:10.1145/3009837.3009857

[27] Roy Margalit and Ori Lahav. 2021. Verifying observational robustness against a c11-style memory model. In *POPL*. 4:1–4:33. doi:10.1145/3434285

[28] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2025. A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries. In *ICSE*. IEEE, 2075–2086. doi:10.1109/ICSE55347.2025.00167

[29] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *PACMPL* 3, POPL (2019), 67:1–67:32. doi:10.1145/3290380

[30] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: Elaborating the de facto standards. In *PLDI*. 1–15. https://doi.org/10.1145/2908080.2908081

[31] Nikolaos S. Papaspyrou. 2001. Denotational semantics of ANSI C. *Comput. Stand. Interfaces* 23, 3 (2001), 169–185. doi:10.1016/S0920-5489(01)00059-9

[32] The Chromium Projects. Last accessed July 9th, 2025. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/.

[33] Rain. Last accessed July 10th, 2025. cargo-nextest. https://github.com/nextest-rs/nextest.

[34] Runtime Verification. Last accessed July 10th, 2025. RV-Match. https://runtimeverification.com/match.

[35] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. ACM, 175–186. doi:10.1145/1993498.1993520

[36] Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S. Pasareanu. 2024. Crabtree: Rust API Test Synthesis Guided by Coverage and Type. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 618–647.

[37] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Pasareanu. 2021. SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In *PLDI*. ACM, 899–913. doi:10.1145/3453483.3454084

[38] MSRC Team. 2019. A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[39] The Rust Team. 2020. The Rust programming language. https://rust-lang.org.

[40] TrustInSoft. Last accessed July 10th, 2025. TrustInSoft Analyzer. https://www.trust-in-soft.com/trustinsoft-analyzer.

[41] TrustInSoft. Last changed 2016. TIS Interpreter (in the Software Heritage Archive). https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/TrustInSoft/tis-interpreter.

[42] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. doi:10.1145/3735592

[43] Qian Wang and Ralf Jung. 2024. Rustlantis: Randomized Differential Testing of the Rust Compiler. *Proc. ACM Program. Lang.* 8, OOPSLA (2024), 1955–1981. doi:10.1145/3689780

Received 2025-07-10; accepted 2025-11-06