RustMC: Automated Verification of Real-World Concurrent Rust *

Oliver Pearce, Julien Lange, and Dan O'Keeffe

Royal Holloway, University of London, UK

Abstract. RustMC is a stateless model checker that targets push-button verification of unmodified concurrent Rust programs. As both Rust and C/C++ compile to LLVM IR, RustMC builds on GenMC which provides a verification framework for LLVM IR. This allows verification of Rust code and any foreign function dependencies. In this tool paper we present key implementation challenges, use-cases, and an empirical evaluation of RustMC. The challenges include intercepting threading operations, promoting memory intrinsics, and handling uninitialised accesses, all stemming from Rust's unique compilation strategy. Case studies adapted from real-world programs show RustMC finds concurrency bugs in unsafe Rust code, FFI calls to C/C++, and atomic operations. We evaluate our tool's functionality on the Loom model checker test suite and a set of production-level Rust concurrent data structures.

Keywords: Model Checking · Rust · LLVM.

1 Introduction

Rust is a popular systems programming language that enforces memory and thread safety through its ownership and borrowing system. Values in Rust have a single owner and can be borrowed either mutably or immutably, with mutable borrowing preventing any concurrent borrows of the same value. These constraints effectively prevent data races in most safe Rust code, as threads cannot make concurrent unsynchronised memory accesses when any access is a write. While Rust's type system prevents many of the common memory safety issues found in other systems programming languages, verification of concurrent programs remains an ongoing challenge. Concurrency bugs occur in Rust for a range of reasons, including atomicity violations, "unsafe" Rust code and interactions between Rust and C/C++ foreign function interface (FFI) dependencies.

In this tool paper, we extend GenMC [17], a stateless model checker that systematically explores executions of concurrent C/C++ programs under weak memory models at the LLVM IR level, to support the verification of multi-threaded Rust programs. By leveraging the LLVM compiler infrastructure's intermediate representation (IR) we derive verifiable LLVM IR modules from both Rust programs and their C/C++ dependencies. RustMC is, to the best of our

* Artefacts available in https://doi.org/10.5281/zenodo.18806668.

knowledge, the only available framework capable of exhaustively exploring the state space of Rust programs and their dependencies, facilitating the detection of data races stemming from Rust, C/C++ and interactions between the languages. While our LLVM IR based approach allows us to verify mixed-language programs, relying on this low-level representation introduces challenges that impact on both RustMC's out-of-the-box support for real-world Rust code and the performance of verification. Specifically, transforming an IR module emitted by the Rust compiler into a verifiable representation requires significant changes to GenMC's compilation and transformation stages. Below we describe how we addressed these challenges and demonstrate the tool's applicability to real-world concurrent programs. The key contributions of this paper are:

1. We present RustMC, an extension of GenMC, the first model checker capable of verifying concurrent Rust programs and their FFI dependencies;
2. We develop novel LLVM transformations to handle Rust-specific challenges, including LLVM memory intrinsics, mixed-size accesses and uninitialised memory accesses, that arise from Rust's unique compilation strategy;
3. We demonstrate RustMC's effectiveness through examples adapted from real-world programs, showing its ability to detect data races in both pure Rust code and mixed-language programs that use C/C++ dependencies;
4. We evaluate RustMC's scalability and support for push-button verification of real-world Rust code. We evaluate our tool on two sets of examples. One consists of 72 tests adapted from the test suite of the Loom model checker [36], the other is based on tests from the 14 most popular crates providing concurrent data structures in Rust; and
5. We provide RustMC as an open-source tool that extends GenMC and enables the fully automated verification of concurrent Rust applications. Our artefact will consist of: the source code of RustMC, its documentation, and the datasets used in our evaluation. All are readily available on GitHub. A video demonstrating the tool is also provided [35].

2 Background

2.1 Concurrency Bugs in Rust

While Rust code provides strong safety guarantees, concurrency bugs can still occur. One potential source is seemingly safe Rust code that harbours data races due to unsafe operations in foreign function dependencies. For example, consider the simple Rust program shown in Figure 1, which calls a C function `inc_C_counter()` from multiple threads. Rust's type system normally prevents concurrent access to shared data. However, the C function bypasses these safeguards and directly manipulates a global counter variable without synchronisation. Rust's borrow checker cannot verify memory safety guarantees across language boundaries. RustMC successfully identifies the race condition automatically, detecting a data race in which one thread attempts to read the `counter` while another thread simultaneously writes to it. Verification is performed au-

```rust
// Rust code
extern "C" {fn inc_C_counter();}
#[test]
fn test_C_counter() {
        thread::spawn(|| {unsafe {inc_C_counter();}});
        thread::spawn(|| {unsafe {inc_C_counter();}});
}
```

```c
// C dependency
int counter = 0;
void inc_C_counter() { counter++; }
```

Fig. 1: A data race caused by an external, non-thread-safe, C function.

tomatically and it is not required to rewrite code using a specific API, as is the case with the Loom model checker [36].

Even safe Rust code can experience atomicity violations when using atomic types, as these types permit shared access and modification across multiple threads [28]. Figure 2 presents code adapted from an atomicity violation bug reported in the `rand` crate [30] (also documented in a study of Rust concurrent programming bugs [40]). Function `is_getrand_available()` implements a common pattern to cache the result of an expensive system call (`getrandom`) in a thread-safe way without using a (more expensive) mutex.

Two `AtomicI64` variables are used to synchronise operations: `CHECKED` indicates whether `getrandom` has been called before and `AVAILABLE` denotes the availability of `getrandom`; both are first initialised to 0. At compile time or run time, the atomic operations at lines 8 and 9 may be re-ordered such that one thread loads `AVAILABLE` before it is set by another.

We show an example of an undesirable interleaving below, assuming threads `t1` and `t2` are concurrently executing `is_getrand_available()` in Figure 2.

t1	t2
`CHECKED.store(...)` (line 9)	`CHECKED.load(...)` (line 4)
`AVAILABLE.store(...)` (line 8)	`AVAILABLE.load(...)` (line 11)

(1)

Here, `t1` encountered `CHECKED==false`, and sets both `CHECKED` and `AVAILABLE` to `true`. From `t2`'s point-of-view, if `t1` has set `CHECKED` before setting `AVAILABLE`, then the function incorrectly returns false for `t2`. To identify this bug, we must specify the property we are interested in with an assert (line 19). This assertion characterises the consistency of concurrent calls to this function (i.e., all threads should see the same results, for all interleavings). RustMC can detect the bug and reports a trace, see (1), violating atomicity. The `unsafe` keyword disables the Rust compiler's safety checks to provide developers with greater flexibility.

```
static CHECKED: AtomicI64 = AtomicI64::new(0);
static AVAILABLE: AtomicI64 = AtomicI64::new(0);
fn is_getrand_available() -> i64 {
    if (CHECKED.load(Ordering::Relaxed) == 0 ){
        let mut buf: [u64; 0] = [];
        let result = getrandom(&mut buf);
        let available = if result == -1 {...} else { 1 };
        AVAILABLE.store(available, Ordering::Relaxed);
        CHECKED.store(1, Ordering::Relaxed);
        available
    } else { AVAILABLE.load(Ordering::Relaxed) }
}
fn main() {
    let t1 = thread::spawn(||{is_getrand_available()});
    let t2 = thread::spawn(||{is_getrand_available()});
    let r1 = t1.join().unwrap();
    let r2 = t2.join().unwrap();

    assert_eq!(r1, r2);
}
```

Fig. 2: An atomicity violation in safe code, adapted from the Rand crate [14].

This can lead to data races in Rust code without any FFI calls. We show further real-world examples of concurrency bugs brought about by the `unsafe` keyword in Appendix A and online [35].

2.2 GenMC: A Stateless Model Checker for C/C++

GenMC [17] supports verification under several weak memory models (e.g., RC11, IMM, LKMM) and represents program behaviours through execution graphs comprising events (nodes) that correspond to individual memory accesses. The tool's architecture consists of three main stages: first, it compiles source code into LLVM IR (e.g. using `clang`); second, it transforms this code to make it easier and faster to analyse, e.g., bounding infinite loops, eliminating dead allocations, etc.; and finally, it conducts verification by exploring program executions and checking for errors such as data races, assertion violations, and memory safety issues. Some of the latest development of the tool includes an optimal dynamic partial order reduction (DPOR) algorithm that explores all possible executions up to some equivalence relation while avoiding redundant explorations [15]. GenMC explores exactly one execution per equivalence class while maintaining polynomial memory requirements.

GenMC provides an ideal foundation for extending stateless model checking to Rust programs. Its integration with LLVM's intermediate representation

creates a natural bridge for verifying Rust code, as both Rust and C/C++ compile to LLVM IR. This alignment enables RustMC to verify not only pure Rust programs but also their interactions with C/C++ dependencies. Moreover, GenMC's three-stage architecture provides a modular framework that RustMC can extend to handle Rust-specific challenges. As GenMC remains under active development, RustMC stands to benefit automatically from future enhancements to the underlying verification framework without requiring significant modifications to its Rust-specific components. Notably, support for mixed-size accesses was recently added as part of the Mixer extension to GenMC [23]. As we discuss in the next section, mixed-size access support is essential for push-button verification of many real-world Rust programs.

3 RustMC: Overview and Implementation Challenges

We next present RustMC, an extension to the GenMC framework for verifying Rust applications. Figure 3 gives an overview of the framework. As in GenMC, there are three stages: compilation, transformation and verification. RustMC also includes an automated test harness component (not shown) that enables a Rust crate's existing test suite to serve as input drivers for verification.

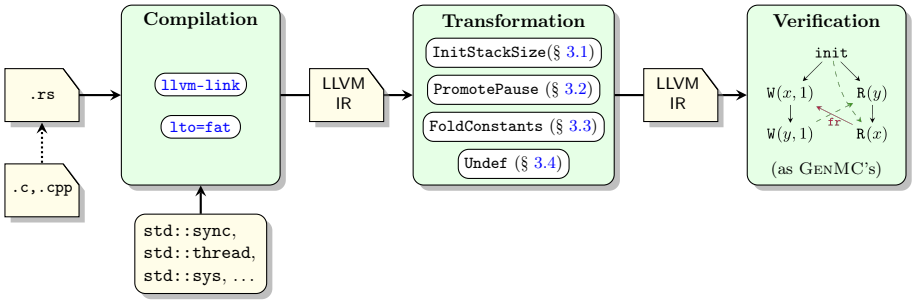

Fig. 3: Architecture of RustMC.

RustMC first compiles the application to LLVM IR, then the `llvm-link` tool [21] is used to link the IR file with the GenMC verification component. This allows for the interception and tracking of relevant threading events. If the application contains FFI calls to C/C++ code, the C IR is also linked. RustMC transforms the resulting IR using a series of LLVM passes to prepare it for verification. In comparison to GenMC, RustMC introduces additional passes that address implementation challenges relating to uninitialised memory accesses (see §3.4), calls to external functions (see §3.1) and unsupported LLVM intrinsics (see §3.2).

Finally, RustMC feeds the transformed IR file to the verification component. This executes the GenMC stateless model checking algorithm to detect any data races or assertion violations across all potential thread interleavings.

3.1 Externally Linked Dependencies

To control a program's schedule, a model checker must intercept thread creation and synchronisation operations. In C/C++, these operations appear directly in a program's compiled IR, allowing GenMC to redefine the corresponding symbols (e.g. `pthread_create`) with its own internal wrapper functions (e.g. `__VERIFIER_thread_create`). However, for Rust applications, many standard library functions are externally linked. As a result, thread creation and synchronisation operations are not visible in the compiled IR.

A naive solution to this problem is to build the entire standard library alongside the application to expose threading calls. However, this approach has several downsides: first, the larger codebase significantly increases the engineering effort required to handle edge cases and unsupported LLVM behaviours; second, processing the much larger IR through each LLVM transformation pass becomes significantly slower; and third, the increased code complexity breaks LLVM optimisation passes that GenMC relies on.

Instead RustMC leverages the `lto=fat` compilation flag to perform link-time optimisation across the target crate and its dependencies, pulling standard library code into standalone LLVM IR modules [8]. This targeted approach keeps the IR size manageable and avoids most LLVM transformation issues. One issue is that the standard library's thread creation implementation queries environment variables and calls native threading internals to set a stack size for new threads. Since GenMC's interpreter does not model environment variables or the required native internals, calls to `thread::spawn` result in a segmentation fault. To mitigate this issue we use an LLVM pass to preinitialise the stack size of new threads to the default value for Rust on Linux.

3.2 Handling LLVM Intrinsics

LLVM supports a number of memory-related intrinsic functions such as those corresponding to the C standard library functions `memcpy()`, `memmove()`, and `memset()`. A call to one of these intrinsics may perform a series of byte-wise accesses. For example, `memcpy(ptr %dest, ptr %src, i64 24)` will copy 24 bytes from `%src` to `%dest`. In GenMC, a custom LLVM pass promotes any intrinsic calls with lengths evaluable at compile time into a series of typed loads and stores. This allows the verification component to handle them as individual accesses. Unpromotable memory intrinsics remain in the IR and trigger a runtime error if they are executed during verification. Rust's compilation strategy frequently produces intrinsics with lengths that cannot be resolved at compile time, necessitating a runtime fallback.

An IR-level implementation would require the use of integer types, as LLVM does not have a representation for raw memory values. This introduces two semantic issues. First, integer types do not carry provenance information, so provenance is lost during copies. This can cause alias analysis to miss pointer escapes, potentially affecting any transformation passes that reason about aliasing. Second, manual implementations of `memcpy` may over-propagate `poison`.

For example, if a `memcpy` is implemented using an `i64` load and some bits within the `i64` are `poison`, the load returns poison for the entire `i64` [24]. By handling memory intrinsics inside the interpreter as direct byte-by-byte copies, RustMC avoids both problems.

This change is sound because the intrinsic implementations are only invoked at runtime by the interpreter, after the `IntrinsicLowering` pass has already converted intrinsic calls into external function calls. Passes earlier in the pipeline may rely on `@llvm.memcpy`'s intrinsic semantics, which guarantee that the source and destination are either equal or non-overlapping. This guarantee would be lost in any IR-level implementation since passes would observe an ordinary function call with no non-overlapping constraint. Note that GenMC's custom memory intrinsic promotion pass also runs after the `IntrinsicLowering` pass, preserving `memcpy` semantics for any prior passes which rely on alias analysis. The Mixer [23] extension to GenMC is used to support any subsequent non-bytewise accesses to memory written bytewise by the interpreter fallback. To reduce the performance overhead associated with mixed-size accesses we run the static promotion pass first, transforming as many intrinsics as possible into loads and stores which match the size of subsequent accesses.

In addition to memory intrinsics, we encountered calls to the `sse2.pause` intrinsic for optimising the performance of spin loops. As this intrinsic is not supported by GenMC's interpreter and does not affect verification we introduce a pass to remove any calls to it.

3.3 Aggregate Types

The Rust compiler emits many values of LLVM aggregate type (e.g. structs, arrays) and frequently uses them in patterns not supported by the verification component, presenting two problems.

First, the compiler can emit statically evaluable `extractvalue` instructions to extract an element from an aggregate. However, the interpreter does not support the direct conversion of constant aggregates into its internal value representation, which is common in IR derived from Rust programs. We transform these `extractvalue` instructions into constants with a custom `FoldInterpreterUnsupportedConstants` pass based on LLVM's `ConstantFoldExtractValueInstruction()` function. Figure 4 illustrates an example of this transformation for LLVM IR that extracts the second field from a constant struct with two `i64` fields. The pass transforms the IR to a simple store of the constant field value (`i64 10`).

Second, while shared globals are initialised using native LLVM functions, LLVM delegates the initialisation of thread-local globals to the client application. GenMC stores per-thread copies of thread-local globals in a dedicated hashmap and initialises them using the LLVM `getConstantValue()` function, which does not support aggregate types with concrete initialisers. The Rust compiler frequently emits thread-local globals of aggregate type, causing the interpreter to crash when initialising them. RustMC addresses this by decomposing thread-local globals into their constituent bytes during initialisation, reconstructing

```
%1 = extractvalue { i64, i64 } { i64 0, i64 10 }, 1
store i64 %1, ptr %6, align 8
```

```
store i64 10, ptr %6
```

Fig. 4: FoldInterpreterUnsupportedConstants transformation of LLVM IR to extract elements from a constant struct (top) to a simple store (bottom).

them on access. As only shared memory accesses are added to the event graph, this change does not affect the execution graphs produced during verification.

3.4 Undefined Values

LLVM supports undefined (`undef`) values to represent indeterminate values. Each time an `undef` value is used by an instruction a different bit pattern may be observed. Typically, `undef` models loading from uninitialised memory [18,22]. This is permitted as long as the `undef` value is not used in a subsequent operation that may trigger immediate undefined behaviour for some potential bit pattern [20]. For example, dividing any value by an `undef` value leads to undefined behaviour as the `undef` value could be 0. GenMC's model of memory accesses does not handle uninitialised loads (it requires that each read has a preceding write to take its value from). As loads from uninitialised memory have no preceding write the tool is unable to construct an execution graph.

```
pub fn foo() -> Result<(), i64> { Ok(()) }
```

```
define { i64, i64 } @foo() unnamed_addr #0 !dbg !7 {
  %_0 = alloca [16 x i8], align 8 ; allocate 16 bytes
  store i64 0, ptr %_0, align 8, !dbg !35 ; put 0 in 1st chunk
  %0 = load i64, ptr %_0, align 8, !dbg !36 ; initialised read
  %1 = getelementptr inbounds i8, ptr %_0, i64 8, !dbg !36
  %2 = load i64, ptr %1, align 8, !dbg !36 ; uninitialised read
  %3 = insertvalue { i64, i64 } poison, i64 %0, 0, !dbg !36
  %4 = insertvalue { i64, i64 } %3, i64 %2, 1, !dbg !36
  ret { i64, i64 } %4, !dbg !36
}
```

Fig. 5: Uninitialised read in Rust (top) and corresponding LLVM IR (bottom).

Figure 5 shows an uninitialised read arising from typical Rust code. The function `foo()` returns the `Ok(T)` variant of `Result<T, E>`. A `Result<T, E>` is stored in memory as a tagged union whose size is determined by its largest

variant. When `T` is the zero-sized unit type `()`, the `Ok()` variant occupies less space than is allocated for the union. The remaining bytes are never written, yet must be loaded when `Result<(),i64>` is read. This occurs at line 6 in Figure 5. Nothing is stored in the second 8-byte chunk of allocated memory, so the load yields `undef`. The function returns `{0, undef}` where `0` is the discriminant for the `Ok()` variant.

To handle uninitialised loads while meeting RustMC's verification requirements, we implement an LLVM pass that explicitly writes undefined values into each stack allocation. This approach allows us to verify programs containing uninitialised reads as RustMC interprets an explicit "`store undef`" as storing 0 to a memory location. Consequently each uninitialised read has a preceding write, as required. Interpreting accesses to uninitialised memory as 0 preserves program semantics provided there is no immediate undefined behaviour arising from uninitialised values taking non-zero bit patterns [20]. We consider this an acceptable trade-off for concurrency verification given the tool's focus on detecting data races and memory ordering issues rather than detecting all forms of undefined behaviour. Alternatives, such as exploring all possible `undef` values, would cause state explosion and make the verification of real-world code impractical. We note that RustMC's undefined value transformation currently only supports stack allocations. Extending it to handle undefined values in dynamic heap allocations using a similar approach should be possible, but is currently unsupported by the tool.

3.5 Soundness, completeness and optimality

As GenMC's soundness, completeness and optimality guarantees depend on preserving the semantics of a program under test, the LLVM transformations introduced by RustMC are designed to be semantics-preserving. Our undefined value handling explores all execution traces in which `undef` is consistently observed as 0 in programs that misuse uninitialised memory. For all other programs, all possible execution traces described by the LLVM semantics are explored.

While promoting memory intrinsics into sequences of typed loads and stores and handling any non-constant intrinsics with a custom internal implementation can increase the number of events in execution graphs — and consequently the number of potential interleavings — this does not compromise GenMC's optimality. The theoretical properties of GenMC's verification algorithm [15] are defined at the level of execution graphs and their exploration; RustMC inherits these properties directly as it reuses GenMC's verification component without modification. Thus, RustMC maintains the same formal guarantees as GenMC while extending support to Rust-specific compilation patterns.

We note that RustMC employs the Mixer algorithm of GenMC to handle mixed-size accesses. Although Mixer is parametric in its choice of memory model, the current implementation is restricted to the Release-Acquire (RA) fragment of RC11.

4 Evaluation

We evaluate RustMC through two complementary experiments that measure its correctness on established benchmarks and its applicability to production-level concurrent data structures.

The first experiment assesses our tool's bug-detection accuracy and practical efficiency on 72 concurrent tests [37] adapted from the Loom model checker test suite. The second experiment evaluates RustMC's support for real-world Rust language features by verifying test functions from 14 popular concurrency-focused crates.

Both experiments use functions marked with the `#[test]` attribute as self-contained verification harnesses. For each test we explore all concurrent behaviours and consider verification successful if RustMC's output matches the expected behaviour. A test's expected behaviour is considered to be a panic when annotated with the `#[should_panic]` attribute, or when its body exercises Rust's unwinding functionality. Because RustMC uses the `panic=abort` strategy, tests that rely on unwinding to recover from a panic will instead abort. We accept a panic outcome as a successful verification for these tests.

4.1 Experiment 1: RustMC and the Loom Test Suite

To evaluate RustMC's effectiveness on established benchmarks, we assessed its performance on 72 concurrent Rust tests adapted from the Loom model checker test suite [37]. These tests cover a range of concurrency primitives including atomics, mutexes, read-write locks, condition variables, thread-local storage, and `UnsafeCell<T>` (primitive for interior mutability in Rust) operations. This constitutes a demanding evaluation: the tests were designed for a production-level model checker operating at the Rust API level, while our research tool works at the lower LLVM IR abstraction, where the same patterns manifest with significantly greater implementation complexity.

For each test, we record whether RustMC's outcome (success, panic, or crash) matches the expected behaviour. We also measure resource consumption: the number of executions explored, maximum memory usage, and CPU time (broken down into the transformation and verification phases). This evaluation aims to measure RustMC's bug-detection accuracy (false negatives and false positives) and its practical efficiency on real-world concurrency scenarios.

Loom requires tests to be written using the Loom API, hence they require adaptation to be used by a push-button model checker like RustMC. Notably if one wants to model check program `P` with Loom, then one must wrap it around the construct `loom::model(|| { P })` which will tell Loom to explore all possible interleavings of `P`. All tests used in this experiment require this wrapper to be removed. In addition, some Loom-specific calls must be re-interpreted (see comment on `spec` below).

Out of the 20 test files in the Loom repository, we rule out three as they either focus on higher-level concurrency issues or primitives we do not target (e.g., channels, memory leak, deadlocks) or aim at testing the Loom API itself. In

Table 1: RustMC verification results on tests adapted from Loom. Status: S: success, P: panic, C: crash. Expected: ✓: matches expected, ×: does not match. Executions: executions explored, Mem: max memory (MB), CPU: user time (s), P.Mem/P.CPU: Transformation pipeline max memory (MB) and user time (s). † indicates a test flagged with `#[ignore]` in the Loom test suite.

File	Test	Status	Expected	Executions	Mem	CPU	P.Mem	P.CPU
arc	basic_usage	S	✓	4	155.6	0.86	179.9	2.09
	sync_in_drop	S	✓	3	153.0	0.83		
	try_unwrap_succeeds	S	✓	1	145.8	0.51		
	try_unwrap_fails	S	✓	1	145.8	0.41		
	try_unwrap_multithreaded	C	–	–	148.0	0.48		
atomic	compare_and_swap_reads_old†	S	×	4	155.4	1.48	179.3	2.23
	fetch_add_atomic	S	✓	2	151.3	0.72		
atomic_int	xor	S	✓	1	136.2	0.47	169.8	1.93
	max	S	✓	1	136.6	0.37		
	min	S	✓	1	136.9	0.47		
	compare_exchange	S	✓	1	136.6	0.38		
	compare_exchange_weak†	S	✓	1	136.8	0.37		
	fetch_update	S	✓	1	137.0	0.34		
atomic_relaxed	compare_and_swap	S	✓	1	150.5	0.56	181.7	2.09
	check_ordering_valid	S	✓	5	157.6	1.09		
	check_ordering_invalid_1	P	✓	10	163.5	1.86		
	check_ordering_invalid_2	P	✓	10	163.0	1.85		
condvar	notify_one	C	–	–	147.9	0.57	179.8	2.12
	notify_all	C	–	–	152.4	1.13		
fence	fence_sw_base	S	✓	2	159.0	1.06	187.9	2.14
	fence_sw_collapsed_store	S	✓	2	159.2	1.00		
	fence_sw_collapsed_load	S	✓	2	159.4	0.88		
	sb_fences	P	×	2	158.8	0.95		
	fence_hazard_pointer	P	×	1	159.1	0.89		
	rwc_syncs	P	×	4	170.5	2.87		
	w_rwc	P	×	3	166.9	2.80		
futures	atomic_waker_valid	C	–	–	149.2	0.46	179.5	2.28
	spurious_poll	S	✓	2	151.4	0.73		
litmus	load_buffering†	S	✓	3	150.8	0.86	178.0	2.05
	store_buffering	S	✓	4	152.2	1.32		
mutex	mutex_enforces_mutual_exclusion	S	✓	1	151.5	0.57	182.0	2.31
	mutex_establishes_seq_cst	S	✓	15	163.3	2.02		
	mutex_into_inner	S	✓	1	151.6	0.76		
rwlock	rwlock_read_one	S	✓	1	152.8	0.59	183.0	2.55
	rwlock_read_two_write_one	S	✓	2	153.2	0.65		
	rwlock_write_three	S	✓	2	154.3	0.62		
	rwlock_write_then_try_write	S	✓	1	148.4	0.39		
	rwlock_write_then_try_read	S	✓	1	148.7	0.44		
	rwlock_read_then_try_write	S	✓	1	148.9	0.46		
	rwlock_try_read	S	✓	1	147.9	0.42		
	rwlock_write	S	✓	1	147.8	0.40		
	rwlock_try_write	S	✓	1	148.0	0.42		
	rwlock_into_inner	S	✓	1	151.8	0.56		
rwlock_r1	rwlock_two_writers	S	✓	24	176.2	11.43	179.2	2.09
smoke*	checks_fail	P	✓	0	170.0	5.74	179.8	2.22
spec	relaxed†	S	×	15	174.8	8.98	186.3	2.30
	acq_rel	S	✓	178	232.9	239.68		
	test_seq_cst†	S	✓	178	233.2	241.92		
thread_api	initial_thread	S	✓	1	192.7	0.61	261.0	3.44
	many_joins	S	✓	1	196.5	0.73		
	alt_join	S	✓	3	207.3	4.73		
	threads_have_unique_ids	C	–	–	202.9	3.00		
	thread_names	C	–	–	202.8	2.87		
	thread_stack_size	S	✓	1	234.5	58.80		
	park_unpark	S	✓	1	192.7	0.67		
thread_local	thread_local_test	S	✓	1	152.1	1.00	178.9	2.12
	nested_with	S	✓	1	143.9	0.36		
	drop_test	S	✓	1	151.6	1.03		
unsafe_cell	atomic_causality_success	S	✓	2	165.9	0.85	202.5	2.60
	atomic_causality_fail	P	✓	0	163.3	0.61		
	unsafe_cell_race_mut_mut_1	P	✓	0	163.5	0.61		
	unsafe_cell_race_mut_mut_2	P	✓	0	168.8	1.34		
	unsafe_cell_race_mut_immut_1	P	✓	0	163.8	0.65		
	unsafe_cell_race_mut_immut_2	P	✓	0	163.6	0.61		
	unsafe_cell_race_mut_immut_3	P	✓	0	168.3	0.95		
	unsafe_cell_race_mut_immut_4	P	✓	0	168.0	0.98		
	unsafe_cell_race_mut_immut_5	P	✓	0	168.9	0.99		
	unsafe_cell_ok_1	S	✓	1	163.8	0.62		
	unsafe_cell_ok_2	S	✓	1	163.8	0.70		
	unsafe_cell_ok_3	S	✓	1	167.0	0.90		
	unsafe_cell_access_after_sync	P	✓	5	169.7	1.13		
yield	yield_completes	S	✓	4	152.2	0.76	175.3	2.02

the remaining 17 files, shown in the first column of Table 1, we additionally omit three further tests aimed at testing the Loom API. For all files, we run RustMC with the GenMC flag `-unroll=1` except for `smoke` where it is necessary to unroll to 3 to detect the bug (highlighted with $*$).

Table 1 gives the results of our experiment. Out of the 72 tests, our tool currently supports 66 tests, it crashes for the remaining 6 ($\approx 9.0\%$). Out of these 66 supported tests, 60 ($\approx 91.0\%$) returned the expected outcome. Out of the 6 misdiagnosed tests, two are not supported by Mixer's memory model, nor by Loom, i.e., Loom returns the wrong result too (they are flagged as `#[ignore]`) because it does not fully model `Ordering::Relaxed`. Similarly, the four unexpected panics in `fences` are due to sequentially-consistent fences which we do not support due to the current limitation of Mixer to the RA memory model. The 6 crashes are due to an external function with variable arguments (i.e., `syscall`). Resource consumption is reasonable, with an average memory footprint of 162.35 MB and mean CPU time of 8.75 seconds per test, though two tests in the `spec` file require significantly more resources (178 executions, 233 MB, 240s). These tests involve 4 threads looping on a `load()` and invoking `thread::yield_now()` in the body of the loop. The original tests include a Loom preemption bound of one (i.e., `builder.preemption_bound = Some(1)`) which we approximated by bounding the loop with one.

4.2 Experiment 2: RustMC and Popular Crates

To evaluate RustMC's applicability to real-world concurrent Rust code, we conducted a characterisation study on tests from 14 popular crates in the Rust ecosystem. Our aim is to establish a baseline for push-button verification of real-world Rust code and identify areas for future improvement.

We selected crates by surveying those with the most downloads in the last 90 days from the concurrency category of `crates.io`. Our current focus is on user-facing concurrent data structures, and so we excluded crates that serve primarily as internal building blocks or rely on higher-level concurrency models (e.g., asynchronous runtimes, channels). While asynchronous runtimes are supported by our tool, we leave a systematic evaluation of async crates to future work. Figure 6 gives an overview of the number of tests available for each crate.

Our test-harness component automatically identifies all test functions in a given crate and invokes RustMC on them as targets for verification. Tests were filtered to remove those dependent on external testing frameworks including Loom and Proptest (External testing frameworks in Fig. 6). We also excluded macro-generated test cases (Macros in Fig. 6) as our test harness does not currently support using them as entry points. This filtering process is conservative: for example, tests removed due to Loom dependencies could potentially be adapted (as in Experiment 1), and macro-generated tests represent an engineering challenge rather than a fundamental limitation. All tests are run with the `-unroll=2` GenMC flag. Figure 7 presents verification results across the 14 crates, with tests categorised by outcome.

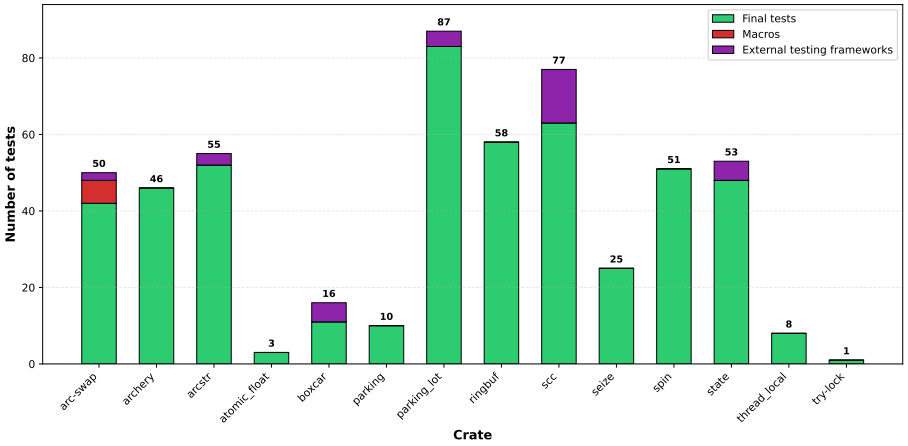

Fig. 6: Selection of tests from popular concurrency-related crates.

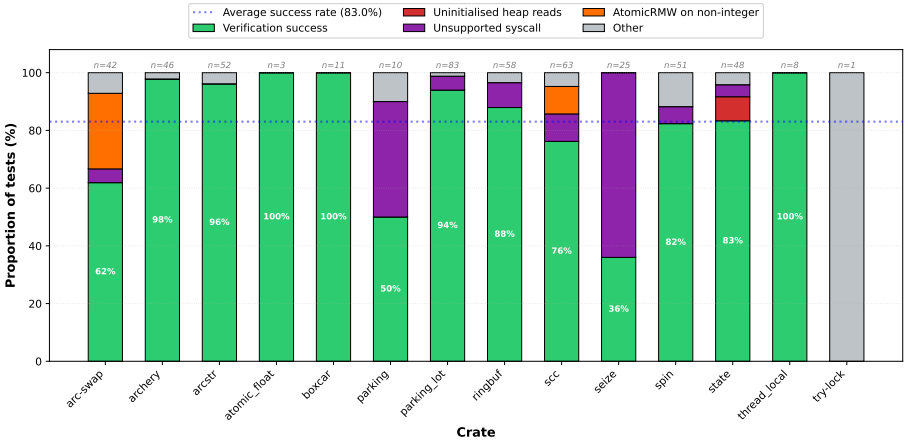

Fig. 7: Results of model-checking tests from popular concurrency-related crates.

RustMC achieved an 83.0% verification success rate over all tests in our dataset. This includes tests from production-grade concurrent data structures such as `archery` and `boxcar`, demonstrating the tool's ability to scale from synthetic benchmarks to complex, real-world concurrent Rust code. Tests that fail to verify fall into three primary categories:

– Uninitialised heap reads: Uninitialised reads from dynamically allocated memory, e.g., when using Rust's `MaybeUninit` type, which requires extending our undefined value handling from stack to heap allocations.
– Unimplemented syscalls: External functions with variable arguments which we do not currently support.

– AtomicRMW on non-integer: Atomic operations on pointers, which are not supported by the verification component.

The Other category in Fig. 7 notably includes (*i*) a few unsupported intrinsics we conservatively convert to runtime errors and (*ii*) four (>1h) timeouts due to unusually large tests [5–7, 39].

The varying success rates across crates (from near-complete verification in some to limited coverage in others) reflects the diversity of concurrent programming patterns in the Rust ecosystem. Some crates heavily utilise language features that align well with RustMC's current capabilities, while others employ patterns that push the boundaries of our LLVM-level analysis. The results provide a baseline for measuring future improvements and identify specific areas (particularly dynamic memory patterns and modelling common syscalls) where extending support would increase coverage of real-world code.

5 Related Work

Rust and its usage have been the focus of much research in the last decade. Notably, RustBelt [12] provides formal verification of Rust's type system and ownership model. Evans et al. [10] shows that while only 30% of Rust libraries explicitly use unsafe code, over half contain unsafe operations somewhere in their dependency chain that bypass Rust's static checks. Astraukas et al. [3] shows that unsafe code appears frequently, particularly for language interoperability, though its unsafe code tends to be straightforward and well-contained.

Dynamic partial order reduction (DPOR) techniques have been successfully applied to model check concurrent programs across different languages and contexts [1, 2, 11]. For Rust specifically, several tools target concurrency bug detection, though with different approaches and tradeoffs compared to ours. Kani [34] is a bounded model checker for Rust which identifies undefined behaviour and verifies custom correctness properties provided by assertions or function contracts for sequential programs.

Miri [13] is part of the Rust compiler project and is advertised as "an undefined behavior detection tool for Rust". It works as an interpreter of Rust's mid-level IR. Miri is particularly effective at detecting undefined behaviours, but its inherently dynamic approach means that it cannot systematically analyse all potential interleavings of concurrent programs. Loom [36] is an adaptation of CDSChecker [25]. It explores interleavings of concurrent code under the C11 memory model using partial-order reduction techniques. In comparison to RustMC/GenMC, the CDSChecker algorithm is less efficient since its backtracking algorithms can explore both redundant and infeasible (prohibited by the memory model) executions [15, 16]. Loom requires developers to port their code to use the Loom API, as opposed to the push-button approach of RustMC. Shuttle [4] is another concurrent code testing tool for Rust, developed at AWS. It is similar to (and inspired by) Loom but aims at better scalability at the cost of soundness. Shuttle uses a randomised scheduler [9] to guarantee good coverage. Lockbud [26, 27] is a static bug detector for blocking bugs and potential

atomicity violations in Rust. It identifies potential atomicity violations by detecting syntactical patterns commonly found in atomicity violation bugs. While this pattern-based approach can identify potential issues, RustMC's systematic exploration of thread interleavings allows it to definitively verify atomicity properties through assert statements, detecting actual violations rather than potential ones. Other work [19, 38] explores verification techniques for Rust's FFI dependencies, but focuses primarily on memory management bugs rather than concurrency issues.

6 Conclusions

RustMC represents a significant step forward in verification of Rust code, and opens new possibilities for detecting subtle concurrency bugs in concurrent Rust programs and their FFI dependencies. Extending GenMC to handle Rust's compilation characteristics presented several technical challenges but its modular architecture allowed us to re-use the back-end verifier. Hence RustMC will automatically benefit from future improvements to GenMC's verification engine.

Looking ahead, RustMC provides a foundation for systematic analysis of real-world concurrent Rust programs, helping developers ensure their code is free from concurrency issues. Our future work will focus on applying RustMC to verify more concurrent code in Rust projects and their dependencies.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 353–367. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_28
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 373–384. ACM (2014). https://doi.org/10.1145/2535838.2535845
3. Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: How do programmers use unsafe Rust? Proc. ACM Program. Lang. **4**(OOPSLA), 136:1–136:27 (2020). https://doi.org/10.1145/3428204
4. AWS: Shuttle, a library for testing concurrent Rust code, url: https://github.com/awslabs/shuttle (Accessed: 2024-1-30)
5. Barretto, J.: Large test in `arc-swap` (mutex fair) (2025), https://github.com/zesterer/spin-rs/blob/91b8001beb553dfe382f65518eef9eb0f1c78602/src/mutex/fair.rs#L598
6. Barretto, J.: Large test in `spin-rs` (mutex spin) (2025), https://github.com/zesterer/spin-rs/blob/91b8001beb553dfe382f65518eef9eb0f1c78602/src/mutex/spin.rs#L454

7. Barretto, J.: Large test in `spin-rs` (mutex ticket) (2025), https://github.com/zesterer/spin-rs/blob/91b8001beb553dfe382f65518eef9eb0f1c78602/src/mutex/ticket.rs#L410
8. rustc book, T.: lto, url: https://doc.rust-lang.org/rustc/codegen-options/index.html#lto (Accessed: 2026-2-19)
9. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: Hoe, J.C., Adve, V.S. (eds.) Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010. pp. 167–178. ACM (2010). https://doi.org/10.1145/1736020.1736040
10. Evans, A.N., Campbell, B., Soffa, M.L.: Is Rust used safely by software developers? In: Rothermel, G., Bae, D. (eds.) ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020. pp. 246–257. ACM (2020). https://doi.org/10.1145/3377811.3380413
11. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 110–121. ACM (2005). https://doi.org/10.1145/1040305.1040315
12. Jung, R., Jourdan, J., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the Rust programming language. Proc. ACM Program. Lang. **2**(POPL), 66:1–66:34 (2018). https://doi.org/10.1145/3158154
13. Jung, R., Kimock, B., Poveda, C., Muñoz, E.S., Scherer, O., Wang, Q.: Miri: Practical undefined behavior detection for rust. Proc. ACM Program. Lang. **10**(POPL) (Jan 2026). https://doi.org/10.1145/3776690
14. klutzy: Fix race condition of atomics (2015), url: https://github.com/rust-random/rand/commit/e0e8263c25dc291f818cd20c034912de5ae05189 (Accessed: 2024-1-30)
15. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. **6**(POPL) (Jan 2022). https://doi.org/10.1145/3498711
16. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 96–110. ACM (2019). https://doi.org/10.1145/3314221.3314609
17. Kokologiannakis, M., Vafeiadis, V.: GenMC: A model checker for weak memory models. In: CAV (1). Lecture Notes in Computer Science, vol. 12759, pp. 427–440. Springer (2021)
18. Lee, J., Kim, Y., Song, Y., Hur, C.K., Das, S., Majnemer, D., Regehr, J., Lopes, N.P.: Taming undefined behavior in llvm. SIGPLAN Not. **52**(6), 633–647 (Jun 2017). https://doi.org/10.1145/3140587.3062343
19. Li, Z., Wang, J., Sun, M., Lui, J.C.S.: Detecting cross-language memory management issues in rust. In: Atluri, V., Pietro, R.D., Jensen, C.D., Meng, W. (eds.) Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13556, pp. 680–700. Springer (2022). https://doi.org/10.1007/978-3-031-17143-7_33
20. LLVM: Llvm ir undefined behavior (ub) manual, url: https://llvm.org/docs/UndefinedBehavior.html (Accessed: 2024-1-30)

21. LLVM: llvm-link - llvm bitcode linker, url: https://llvm.org/docs/CommandGuide/llvm-link.html (Accessed: 2024-1-30)
22. LLVM: 'load' instruction, url: https://llvm.org/docs/LangRef.html#load-instruction (Accessed: 2024-1-30)
23. Marmanis, I., Kokologiannakis, M., Vafeiadis, V.: Model checking C/C++ with mixed-size accesses. Proc. ACM Program. Lang. **9**(POPL), 2232–2252 (2025). https://doi.org/10.1145/3704911
24. Mitenkov, G., Lopes, N., Lee, J.: [RFC] Introducing a byte type to LLVM. LLVM Developer Mailing List (Jun 2021), https://lists.llvm.org/pipermail/llvm-dev/2021-June/150883.html
25. Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. ACM Trans. Program. Lang. Syst. **38**(3), 10:1–10:51 (2016). https://doi.org/10.1145/2806886
26. Qin, B., Chen, Y., Liu, H., Zhang, H., Wen, Q., Song, L., Zhang, Y.: Understanding and detecting real-world safety issues in Rust. IEEE Trans. Software Eng. **50**(6), 1306–1324 (2024). https://doi.org/10.1109/TSE.2024.3380393
27. Qin, B.: lockbud: statically detect memory, concurrency bugs and possible panic locations for rust., https://github.com/BurtonQin/lockbud
28. Reference, T.R.: Interior mutability, url: https://doc.rust-lang.org/reference/interior-mutability.html (Accessed: 2024-1-30)
29. Roundy, D., Contributors, G.: Issue #20: Intern<t>: Data race allowed on t (2021), url: https://github.com/droundy/internment/issues/20 (Accessed: 2024-1-30)
30. Rust-random maintainers: Rand, url: https://github.com/rust-random/rand (Accessed: 2024-1-30)
31. Rustonomicon, T.: Data races and race conditions, url: https://doc.rust-lang.org/nomicon/races.html (Accessed: 2024-1-30)
32. Steve Klabnik, C.N.: Url: https://doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html (Accessed: 2024-1-30)
33. Steve Klabnik, C.N.: Raw pointers, https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html?highlight=raw%20pointer#dereferencing-a-raw-pointer (Accessed: 2024-1-31)
34. Team, T.K.: The kani rust verifier (2025), https://github.com/model-checking/kani
35. The Authors: RustMC, url: https://github.com/Ollie-Pearce/rustmc
36. Tokio: Loom, concurrency permutation testing tool for Rust, url: https://github.com/tokio-rs/loom (Accessed: 2024-1-30)
37. Tokio: Loom, concurrency permutation testing tool for Rust, https://github.com/tokio-rs/loom/tree/master/tests
38. Toman, J., Pernsteiner, S., Torlak, E.: Crust: A bounded verifier for Rust (N). In: Cohen, M.B., Grunske, L., Whalen, M. (eds.) 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. pp. 75–80. IEEE Computer Society (2015). https://doi.org/10.1109/ASE.2015.77
39. Vaner, M.: Large test in `arc-swap` (2025), https://github.com/vorner/arc-swap/blob/19f0d661a27bb6312c6ba9e19e1453db19c30ab5/src/lib.rs#L1169
40. Yu, Z., Song, L., Zhang, Y.: Fearless concurrency? understanding concurrent programming safety in real-world Rust software. CoRR **abs/1902.01906** (2019), http://arxiv.org/abs/1902.01906

A Additional examples

A.1 Out of bounds accesses

Data races in Rust may also cause out of bounds accesses to allocated memory, resulting in an immediate unrecoverable `panic!` error [32]. Figure 8 gives an example of such a bug.

```
1  fn main() {
2      let data = vec![1, 2, 3, 4];
3      let idx = Arc::new(AtomicUsize::new(0));
4      let other_idx = idx.clone();
5
6      thread::spawn(move || {
7          other_idx.fetch_add(10, Ordering::SeqCst);
8      });
9
10     if idx.load(Ordering::SeqCst) < data.len() {
11         unsafe {
12             let i = idx.load(Ordering::SeqCst);
13             let x = *data.get_unchecked(i);
14         }
15     }
16 }
```

Fig. 8: A data race which leads to an unrecoverable `panic!` error, adapted from [31].

Variable `other_idx` is a clone of the `Arc` type used as a thread safe reference counter, i.e., `idx` and `other_idx` both contain references to the same value. Recall that atomics have interior mutability and are not subjected to restrictions on mutable aliasing. An ownership move is performed at line 6, and the thread which has been passed ownership of `other_idx` increments the variable by 10. At line 10, a bounds check is performed and if `idx` is smaller than the length of the allocated `vec`, then the position of `idx` in the allocated `vec` array is accessed. In an interleaving where `other_idx` is incremented after the bounds check at line 10, an out-of-bounds read occurs at line 13.[1] This leads to an immediate `panic!` call which is detected by RustMC along with the interleaving responsible for the error.

[1] Method `get_unchecked()` is an unsafe method that retrieves an element from a vector without performing any bounds checking, i.e., it is a potentially faster version of the regular `get()` method.

A.2 Raw Pointers

In Rust, developers can use raw pointers to bypass the language's safety guarantees around aliasing and mutability when necessary [33]. In [26], Qin et al. show that sharing data between threads through raw pointers to shared memory is a prevalent source of concurrency bugs. Figure 9 illustrates a concrete example of such an issue. While raw pointers can be created in safe code as is seen at line

```
1  fn main() {
2      let mut x: i64 = 2;
3      let x_ptr = &mut x as *mut i64; // Pointer #1 to x
4
5      unsafe {
6        let x_ptr_2: &mut i64 = &mut *x_ptr; // Pointer #2 to x
7        let handle = thread::spawn(move || {
8              *x_ptr_2 = 5;
9          });
10       *x_ptr = 10;
11       handle.join().unwrap();
12     }
13     let y = x;
14 }
```

Fig. 9: A data race caused by the use of raw pointers for sharing data between threads.

3, dereferencing a raw pointer is always considered unsafe and must be marked as such. At line 6, `x_ptr` is dereferenced, creating a second mutable reference to the value held at `x`. Ownership of this reference is transferred to a new thread where the underlying value `x` is mutated. This is performed concurrently with another write to `x` at line 10.

Now `y` may be assigned different values (10 or 5) depending on the observed interleaving. RustMC is able to detect this as a non-atomic race and reports the conflicting write events at lines 8 and 10.

A.3 Data Race in Thread-safe Traits

We give another example that demonstrates how incorrect implementations of Rust's thread safety traits can lead to data races. The `Sync` trait, which marks types as safe to share between threads, requires careful implementation to maintain Rust's safety guarantees. This type of bug has occurred in practice, e.g., the `internment` crate contained a data race due to an unsafe implementation of `Sync` for its `Intern<T>` struct [29]. The implementation allowed the struct to be shared between threads even when instantiated with non-thread-safe types, potentially leading to memory corruption. To illustrate this class of bug, we

```
1  struct MyStruct { data: Cell<i64>, }
2
3  unsafe impl Send for MyStruct {}
4  unsafe impl Sync for MyStruct {}
5
6  impl MyStruct {
7      fn new(value: i64) -> Self {
8          MyStruct { data: Cell::new(value), }
9      }
10     fn increment(&self) {self.data.set(self.data.get() + 1);}
11 }
12
13 fn main() {
14     let foo = Arc::new(MyStruct::new(0));
15     let foo_clone1 = foo.clone();
16     let foo_clone2 = foo.clone();
17     let t1 = thread::spawn(move || {foo_clone1.increment();});
18     foo_clone2.increment();
19     t1.join().unwrap();
20     let final_value = foo.data.get();
21 }
```

Fig. 10: Data race in safe Rust code.

create a minimal example where a struct containing a `Cell<i64>` incorrectly implements `Sync`. The `Cell` type provides interior mutability but is explicitly designed to be non-thread-safe, making this implementation unsound. Figure 10 shows how this type definition can lead to data races.

`MyStruct` contains a single field data which is a `Cell<i64>`, i.e., a type that provides interior mutability (data can be mutated even via an immutable reference). Lines 3 and 4 declare `MyStruct` as safe to send between threads (`Send`) and safe to share references across threads (`Sync`). As they appear in an `unsafe` block, the programmer is taking responsibility for thread safety. However, `MyStruct` implements the `increment()` method, which is clearly not thread-safe. To expose the issue, we provide a `main()` function which makes two concurrent calls to `increment()`. Line 14 wraps a new `MyStruct` in an `Arc` (Atomic Reference Counted), which allows safe sharing between threads. Lines 15 and 16 create new references to `foo`; hence both threads are modifying the same underlying data. When given Figure 10 as input, RustMC correctly identifies the race. It detects a conflict between the unsynchronised accesses to `foo` made by the main thread at line 18 and the spawned thread at line 17.