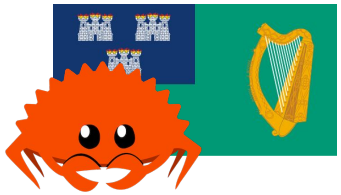
Unsafe and Sound:
Testing Unsafe Rust

Brian Pane <brianp@brianp.net>



Presented to **Rust Dublin** 2026-06-03

Agenda

Share

- tools,
- techniques,
- and best practices

for validating `unsafe` Rust code

Motivation

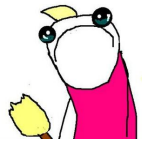
A couple of years ago, I started contributing to open-source Rust projects

- Week 1: Let's make all the critical infrastructure memory-safe!

- Week 2: This is easy! The compiler ensures we'll never have memory corruption!

- Week 3: Wait, all these projects use a lot of `unsafe` code

  … where those compiler guarantees don't apply.



19% of public Rust crates use `unsafe`

(Rust Foundation report, 2024)

Unsafe Rust is Sometimes Inevitable

Calling C libraries

Exposing raw pointers in Rust APIs to be called from C

Inline assembly blocks with `asm!`

Optional CPU features gated with `#[target_feature]`

The Big Challenge: How to Make Unsafe Rust Sound?

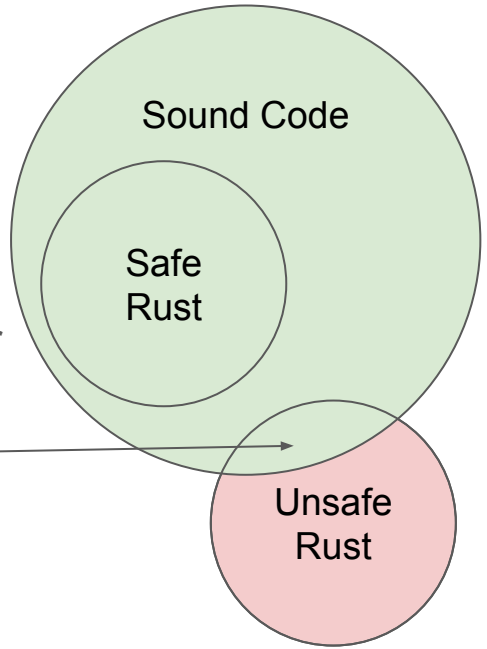
Unsafe Rust is often needed in kernel, networking, encryption, & embedded code

… where we *really* don't want memory-safety errors.

Two distinct concepts in Rust:

- **Sound**: no Undefined Behaviour
- **Safe**: compiler can prove there's no Undefined Behaviour

If the code isn't **safe**, can we still verify that it is **sound**?



Example: Some Unsafe Rust (and C)

```rust
pub unsafe fn write_buffer(addr: *mut u8, size: usize, pattern: u8) {
    let mut dest = addr;
    for _ in 0..size {
        unsafe { *dest = pattern };
        dest = unsafe { dest.add(1) };
    }
}

unsafe extern "C" {
    pub fn write_buffer_c(data: *mut u8, size: size_t, pattern: u8) -> u64;
}
```

```c
extern void write_buffer_c(uint8_t *data, size_t size, uint8_t pattern) {
    size_t i;
    for (i = 0; i < size; i++) {
        *data++ = pattern;
    }
}
```

Example: More Unsafe Rust (and C)

```rust
pub unsafe fn read_buffer(addr: *const u8, size: usize) -> u64 {
    let mut sum = 0;
    let mut dest = addr;
    for _ in 0..size {
        sum += unsafe { *dest } as u64;
        dest = unsafe { dest.add(1) };
    }
    sum
}
```

```c
extern uint64_t read_buffer_c(const uint8_t *data, size_t size) {
    uint64_t sum = 0;
    size_t i;
    for (i = 0; i < size; i++) {
        sum += *data++;
    }
    return sum;
}
```

Example: Some Unsound Test Cases

```
#[test]
fn read_overflow_stack_rust() {
    let buf1 = [0, 1, 2, 3, 4];
    let _buf2 = [0, 0, 0, 0, 0];
    assert_eq!(unsafe { read_buffer(buf1.as_ptr(), 6) }, 10);
}

#[test]
fn write_overflow_heap_c() {
    let mut buf1 = Box::new([0, 1, 2, 3, 4]);
    let _buf2 = Box::new([0, 0, 0, 0, 0]);
    unsafe { write_buffer_c(buf1.as_mut_ptr(), 6, b'!') };
    assert_eq!(*buf1, [b'!'; 5]);
}
```

Example: More Unsound Test Cases

```
fn freed_stack_ref() -> *const u8 {
    let buf = [0, 1, 2, 3, 4];
    buf.as_ptr()
}

#[test]
fn use_after_free_stack_rust() {
    let addr = freed_stack_ref();
    black_box(unsafe { read_buffer(addr, 5) });
}

#[test]
fn use_after_free_heap_c() {
    let buf1 = Box::new([0, 1, 2, 3, 4]);
    let addr = buf1.as_ptr();
    drop(buf1);
    black_box(unsafe { read_buffer_c(addr, 5) });
}
```

Test Matrix

Our example codebase's test suite has 12 intentional memory-safety problems.

	Heap Buffer / from Rust	Heap Buffer / from C	Stack Buffer / from Rust	Stack Buffer / from C
Buffer Read Overflow				
Buffer Write Overflow				
Use After Free				

Tests Pass! 100% Test Coverage! Is the Code Sound?

```
% cargo test
   ...
test tests::read_overflow_stack_c ... ok
test tests::read_overflow_heap_c ... ok
test tests::read_overflow_stack_rust ... ok
test tests::use_after_free_stack_c ... ok
test tests::use_after_free_heap_c ... ok
test tests::use_after_free_stack_rust ... ok
test tests::read_overflow_heap_rust ... ok
test tests::use_after_free_heap_rust ... ok
test tests::write_overflow_heap_c ... ok
test tests::write_overflow_heap_rust ... ok
test tests::write_overflow_stack_c ... ok
test tests::write_overflow_stack_rust ... ok
```

Function Coverage	Line Coverage	Region Coverage
100.00% (15/15)	100.00% (84/84)	100.00% (151/151)
100.00% (15/15)	**100.00% (84/84)**	**100.00% (151/151)**

Static Analysis Tools for Rust

Rudra

+ Research project from Georgia Tech
+ Has found 100+ bugs in open source Rust projects
- No longer maintained
- Doesn’t build on ARM

Kani

+ Open source project developed by AWS
+ Has found bugs in hypervisors & QUIC implementations
- Requires some extra code to guide its analysis
- Doesn’t support C FFI calls

Instrumenting Code for Kani Analysis

```
#[cfg(kani)]
#[kani::proof]
fn check_read_overflow_stack_rust() {
   let buf = [0, 1, 2, 3, 4];
   let size = kani::any();
   black_box(unsafe { read_buffer(buf.as_ptr(),size) });
}
```

Example of Kani Output

```
% cargo kani
  ...
SUMMARY:
 ** 2 of 46 failed
Failed Checks: Offset result and original pointer must point to the same
allocation
 File: "/Users/runner/work/kani/kani/library/kani/src/lib.rs", line 57, in
kani::rustc_intrinsics::offset::<u8, *const u8, usize>
Failed Checks: dereference failure: pointer outside object bounds
 File: "src/lib.rs", line 19, in read_buffer
```

What Did Kani Catch?

	Heap Buffer / Rust	Heap Buffer / C	Stack Buffer / Rust	Stack Buffer / C
Buffer Read Overflow	✔️	❌	✔️	❌
Buffer Write Overflow	✔️	❌	✔️	❌
Use After Free	✔️	❌	✔️	❌

Dynamic Analysis

With static analysis tools for Rust still in their early stages,

… how about using a dynamic analyzer?

Dynamic analysis tools run the code in an emulator or add instrumentation at compile time.

- – Slower
- – Can only catch problems that your test cases exercise
- + But available
- + And powerful

Valgrind

Dynamic analysis tool to find memory safety errors in C/C++ programs

+ Works for Rust too!

- Currently Linux-only (there's a macOS porting effort)

+ Works well on both x86_64 (64-bit x86) and aarch64 (64-bit ARM)

```
env CARGO_TARGET_$(rustc -vV | \
    sed -n 's|host: ||p' | tr '[a-z]-' '[A-Z]_' \
)_RUNNER="valgrind --track-origins=yes --error-exitcode=1" \
cargo test
```

Example of Valgrind Output

```
==5377== Thread 2 tests::read_ove:
==5377== Invalid read of size 1
==5377==    at 0x4016910: read_buffer_c (c_lib.c:8)
==5377==    by 0x4014A0F: unsafe_rust_testing::tests::read_overflow_heap_c
(lib.rs:72)
  ...
==5377==  Address 0x4bbbb25 is 0 bytes after a block of size 5 alloc'd
==5377==    at 0x4955618: malloc (vg_replace_malloc.c:447)
==5377==    by 0x401601F: alloc (library/alloc/src/alloc.rs:95)
==5377==    by 0x401601F: alloc::alloc::Global::alloc_impl_runtime
(library/alloc/src/alloc.rs:190)
==5377==    by 0x40143AF: alloc_impl (library/alloc/src/alloc.rs:312)
==5377==    by 0x40143AF: allocate (library/alloc/src/alloc.rs:429)
==5377==    by 0x40143AF: alloc::boxed::box_new_uninit
(library/alloc/src/boxed.rs:248)
==5377==    by 0x40148CB: new<[u8; 5]> (library/alloc/src/boxed.rs:286)
==5377==    by 0x40148CB: unsafe_rust_testing::tests::read_overflow_heap_c
(lib.rs:70)
```

What Did Valgrind Catch?

	Heap Buffer / Rust	Heap Buffer / C	Stack Buffer / Rust	Stack Buffer / C
Buffer Read Overflow	✔️	✔️	❌	❌
Buffer Write Overflow	✔️	✔️	❌	❌
Use After Free	✔️	✔️	✔️	❌

Miri

Dynamic analysis tool to detect Undefined Behaviour in Rust programs

Runs the rustc+LLVM compiler's Intermediate Representation in an emulator

+ Detects Undefined behaviour in unsafe Rust

+ Works on Windows and macOS and Linux, x86_64 and ARM

– Doesn't support calls to C/C++

```
% MIRIFLAGS="-Zmiri-disable-isolation" cargo \
    +nightly miri test --no-fail-fast
```

Example of Miri Output

```
test tests::read_overflow_stack_rust ... error:Undefined Behavior: memory access
failed: attempting to access 1 byte, but got alloc44134+0x5 which is at or beyond
the end of the allocation of size 5 bytes
  --> src/lib.rs:25:25
   |
25 |         sum += unsafe { *dest } as u64;
   |                         ^^^^^ Undefined Behavior occurred here
   |
   = help: this indicates a bug in the program: it performed an invalid operation,
and caused Undefined Behavior
help: alloc44134 was allocated here:
  --> src/lib.rs:47:13
   |
47 |         let buf1 = [0, 1, 2, 3, 4];
   |             ^^^^
   = note: this is on thread `tests::read_overflow_stack_rust`
   = note: stack backtrace:
           0: read_buffer
               at src/lib.rs:25:25: 25:30
           1: tests::read_overflow_stack_rust
               at src/lib.rs:49:29: 49:58
```

What Did Miri Catch?

	Heap Buffer / Rust	Heap Buffer / C	Stack Buffer / Rust	Stack Buffer / C
Buffer Read Overflow	✔️	❌	✔️	❌
Buffer Write Overflow	✔️	❌	✔️	❌
Use After Free	✔️	❌	✔️	❌

ASAN (Address Sanitizer)

Compiler can inject runtime memory-safety checks into the generated executable

+ Generally lower runtime overhead than Valgrind and Miri
+ Works for both C and Rust
+ Supported on Linux and macOS

```
env ASAN_SYMBOLIZER_PATH=$(which llvm-symbolizer-21) \
    ASAN_OPTIONS=halt_on_error=0 \
    RUSTFLAGS=-Zsanitizer=address \
  cargo +nightly test --features=asan --no-fail-fast
```

Example of ASAN Output

```
==6066==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xf13876de0045 at
pc 0xbe883beef1f8 bp 0xf138779fdc70 sp 0xf138779fdc68
READ of size 1 at 0xf13876de0045 thread T1
    #0 0xbe883beef1f4 in unsafe_rust_testing::read_buffer
/home/brian/code/unsafe-rust-testing/src/lib.rs:19:25
    #1 0xbe883bef2fc4 in unsafe_rust_testing::tests::read_overflow_stack_rust
/home/brian/code/unsafe-rust-testing/src/lib.rs:51:29
  ...
Address 0xf13876de0045 is located in stack of thread T1 at offset 69 in frame
    #0 0xbe883bef2e60 in unsafe_rust_testing::tests::read_overflow_stack_rust
/home/brian/code/unsafe-rust-testing/src/lib.rs:48

  This frame has 2 object(s):
    [32, 40) '_5' (line 51)
    [64, 69) 'buf1' (line 49) <== Memory access at offset 69 overflows this variable
```

What Did ASAN Catch?

	Heap Buffer / Rust	Heap Buffer / C	Stack Buffer / Rust	Stack Buffer / C
Buffer Read Overflow	✔️	✔️	✔️	✔️
Buffer Write Overflow	✔️	✔️	✔️	✔️
Use After Free	✔️	✔️	✔️	✔️

Fuzz Testing

Run your code with random inputs to help trigger edge cases.

- Requires extra test scaffolding
- Nondeterministic
+ The `cargo fuzz` tool helps set up and run the fuzz tests
+ Works on all platforms

```
cargo +nightly fuzz run "$FUZZ_TEST_NAME" \
    -- --max_total_time=10
```

Instrumenting Code for Fuzz Testing

```
#![no_main]

use libfuzzer_sys::fuzz_target;
use unsafe_rust_testing::*;

fuzz_target!(|size: usize| {
    let buf1 = [0, 1, 2, 3, 4];
    let _buf2 = [0, 0, 0, 0, 0];
    assert_eq!(unsafe { read_buffer(buf1.as_ptr(), size) }, 10);
});
```

Example of Fuzz Testing Output

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==65105==ERROR: AddressSanitizer: BUS on unknown address (pc 0x00010496f2a4 bp
0x00016b492110 sp 0x00016b492060 T0)
==65105==The signal is caused by a READ memory access.
==65105==Hint: this fault was caused by a dereference of a high value address (see
register values below).  Disassemble the provided pc to learn which register was
used.
    #0 0x00010496f2a4 in read_buffer_c+0xac (read_heap_c:arm64+0x1000032a4)
    #1 0x00010496eb18 in rust_fuzzer_test_input lib.rs:363
```

Note that `cargo fuzz` ran the tests through ASAN automatically.

What Did Fuzz Testing Catch?

	Heap Buffer / Rust	Heap Buffer / C	Stack Buffer / Rust	Stack Buffer / C
Buffer Read Overflow	✔️	✔️	✔️	✔️
Buffer Write Overflow	✔️	✔️	✔️	✔️

Observations

Lots of tools available…

- Each has drawbacks
  - Incompatibility
  - Extra test code required
  - Slow test execution
  - Nondeterminism
  - False negatives
- But each also discovers important bugs!

Recommendations

Start with a solid foundation

- Peer review
- Thorough regression test suite

Use a combination of multiple techniques…

- Static analysis
  - If it works with your codebase
- Dynamic analysis
  - MIRI if you're using pure Rust
  - ASAN for hybrid Rust + C/C++
- Fuzz testing

… to help ensure that your `unsafe` code is sound.

Q&A

Appendix

What Else Can Valgrind Do?

Encryption code often needs to run in constant time:

- No conditional branches based on secret inputs

  ❌ `if private_key.is_even() { … }`

- No memory/cache addressing based on those secrets, either

  ❌ `hash += lookup_table[private_key % 256]`

Writing constant-time code is tricky

- Compiler optimizations can transform constant-time code into non-constant-time (e.g., CVE-2026-23519)

What Else Can Valgrind Do?

Valgrind has an API to mark blocks of memory as uninitialized at runtime.

- After initializing a secret, call the API to tell Valgrind it's *un*initialized.
- Valgrind will then report an error if that value is used for:
  - Conditional branches
  - Computing addresses for memory reads/writes

Note: There are types of non-constant-time code that this won't catch.

- E.g., what if the CPU's multiply operation runs faster for small inputs?
- But Valgrind still can be a powerful tool in your overall test suite.

Static Analysis: What Can the Compiler Find?

Not much, it turns out. But a few directives can help make the unsafe code more manageable for authors and maintainers:

Constrain the footprint of Rust in your codebase:

```
#![deny(unsafe_code)]  // Override only where needed
```

Require `// SAFETY` explanation comments on unsafe blocks:

```
#![warn(clippy::undocumented_unsafe_blocks)]
```

Require explicit `unsafe { … }` blocks even inside `unsafe` functions:

```
#![forbid(unsafe_op_in_unsafe_fn)]
```