“Rewrite it in Rust” Considered Harmful?
Security Challenges at the C-Rust FFI
Anonymous Authors

ABSTRACT
You must be reading this because you haven’t heard the good news. Rewrite it in Rust! Seriously—rewrite it in Rust!
Stop reading this, go pick any software you use and love (or more likely use but hate)—and rewrite it in Rust.
Have you not tried the Rust cat, grep and find?
Seriously, it will blow your mind.
What? You’re too busy and don’t have time?
It’s okay. Really, it’s fine!
Just rewrite it one piece at a time!
And, in return, your system will be, I promise you, Faster and more secure, through and through.
We’ve never seen a slowdown. Not one bug. Not two.
What? I’m lying? I sound like SBF and Madoff too?
Okay, I guess you’re right. It’s too good to be true.
At the boundary there is lots of glue.
And if you get it wrong, your Rust types are through.
You’re back to C, bits and bytes,
Worse off because of the rewrite.
Lucky for you, we have fancy types to make this right.
And precise C-Rust conditions we can automatically verify. How? Where? Why?
Read on. We’ll clarify.
This is a story about FFI.

1 INTRODUCTION
Responding to the seemingly never-ending series of security threats attributed to memory unsafety—including 60 to 70 percent of today’s browser and kernel vulnerabilities [32, 38]—system developers today are increasingly turning to memory-safe languages. Well, they’re turning to Rust.
This is because Rust promises to be both fast and safe, and targets the sort of abstractions needed for low-level systems implementation, including interaction with the OS, low-level memory management, and concurrency [24]. These advantages and others (in particular, Rust’s tooling support) helped Rust cross the chasm from early adopters to infrastructures companies like Amazon and Google. Even Consumer Reports is on board the RIIR train [21]:

As much as possible, companies, government organizations, and other entities should commit to using memory-safe languages [read: Rust] for new products and tools and newly developed custom components.

Less widely acknowledged, though, is that rewriting only components of a large C/C++ system in a memory-safe language introduces an additional attack surface: the foreign function interface (FFI) boundary between the new component and existing code. In fact, interaction with Rust can make the situation worse.† Consider this C function code:

```c
void add_twice(int *a, int *b) {
  *a += *b;
  *a += *b;
}
```

It’s a bit weird—it performs in-place arithmetic on integer pointers—but also the reason we’d want to rewrite it to safe Rust:

```rust
// [no_mangle]
pub extern "C"
fn add_twice(a: &mut i32, b: &i32) {
  *a += *b;
  *a += *b;
}
```

Unfortunately, Rust makes different assumptions about `a` and `b` and calling, say, `add_twice(&bar, &bar)` from C results in undefined behavior. This is because the Rust compiler can—and does—optimize `add_twice` to `*a += 2 * *b`. (After all, in Rust, `a` and `b` can’t alias.) And, this optimization can introduce a new memory unsafety bug. If the C application uses `add_twice` to update memory-relevant data (e.g., to double the size of a buffer twice)² using the “safe” Rust function is worse than using the original “unsafe” C version.

The striking thing about this example is that both the original C code and the Rust code in isolation pass the respective compiler with no errors. Yet, in tandem the C and Rust code silently invoke undefined behavior which—depending on the architecture, version of Rust, and version of LLVM—can result in an arbitrary memory unsafety.

In practice, this problem is neither artificial nor easily preventable. Fundamentally, Rust and C/C++ cannot interact

†We’d like to think that that Apple is not slow to adopt Rust, but ahead of everyone working on “improved memory handling” and “improved checks” for type confusion for their C — the language not quality — codebase [2].

The security updates are reassuring.

²Sure, this is a contrived example. But we all know there is probably C code out there doing this. Probably in the kernel.
directly—they have different approaches to typing, memory management, and control-flow. The result is manually written "glue" code, which can easily break implicit assumptions (e.g., calling conventions and data representations), critical invariants (e.g., memory- and type-safety, synchronization and resource-handling protocols), and introduce undefined behavior bugs across language boundaries (e.g., unwinding panics, integer representation bugs, silently creating invalid values for enumerated and tagged union types). This problem is widely understood in a broader context (e.g., [1, 23, 29, 35, 37]); FFIIs are notoriously tricky and prone to bugs [7], even in Rust FFI, where unsafe is virtually unavoidable [4, 28]. Developers currently lack principled techniques and tools to develop safe FFIs, so today rewriting code in Rust is likely to introduce new bugs and vulnerabilities [42]—and make existing vulnerabilities easier to exploit [33, 36].

In the rest of this paper, we look at real-world attempts to rewrite components of large C/C++ systems in Rust—and the new class of bugs and issues that developers introduce when writing FFI code (Section 2). The impedance mismatch between Rust and C results in lots of unsafe code at the FFI boundary—and a significant challenge for developers porting components to Rust. This is not surprising—but then again the Venn diagram of system builders who know Rust and those who knew Modula 3 and know the problem deeply [6, 39] has no overlap. With Rust we have another shot—at least if we extend the Rust FFI boundary with a refinement type system we call R3 (Section 3), (Rewrite it in Rust with Refinements). Our type system has two parts: (1) an allocation tracker on the C/C++ side, which tracks C allocations and allows us to enforce needed Rust safety invariants on pointers that are passed to Rust; and (2) a refinement type tracker on the Rust side [30] for verifying Rust code [3, 5, 20, 22], which ties together pointers and their bounds and can be used to track security checks and preconditions for Rust code exposed via the FFI. Together they lower the specification and proof burden endemic to almost all verification tools—and they eliminate FFI bugs much like Rust eliminates memory safety bugs: with zero-overhead but somewhat annoying syntax and error messages that you’ll, over time, love.

2 FFI SAFETY ISSUES

In this section, we investigate the security vulnerabilities that occur when porting C and C++ components to Rust in real world applications. Because we are specifically interested in bugs at the FFI layer, we do not consider bugs that are present in the original C or C++ code and do not directly affect the ported code. That is, we model a scenario where the original code is memory-safe; the ported code is memory-safe; and we consider memory safety and undefined behavior that may arise across the FFI layer between the two pieces of code.

We assume that a developer is acting in good faith to port the code, but may not get all boundary code correct. Due to errors in porting, the buggy application may pass malformed or wrong inputs to FFIs, such as incorrect values for pointers and buffer lengths. Since the memory is shared between the C/C++ application and the Rust library, any incorrect handling of this input from the Rust library could result in memory safety error that affects the entire application.

Ported Libraries and Applications. We analyzed the Rust implementation of two network protocols libraries: the TLS library rustTLS [18] and the HTTP library Hyper [15], and their FFIs [19]. These libraries and their C bindings are actively developed and currently integrated in Curl [13], and thus offer a good case study for C-to-Rust FFIs. We also consider a few other projects: ENCODING.C [14], a Rust implementation of the Encoding standard that replaced a C++ implementation in Firefox; Ockam [12], a secure end-to-end communication library; Artichoke [11], a Rust implementation of the Ruby language; and core challenges found by the Rust language team [16, 17];

We categorize these issues in Section 2 as follows. First, we have violations of spatial and temporal memory safety. Second, we find a common class of errors in exception safety. Unwinding the stack across the FFI boundary is undefined behavior, and so constitutes a serious failure case which has sometimes gone unnoticed. Third, we discuss errors related to type safety and Rust critical invariants, including aliasing, pointer safety assumptions, and reference mutability. Finally, we discuss other miscellaneous kinds of undefined behavior.

2.1 Spatial and Temporal Safety

Rust, C, and C++ have fundamentally different approaches to memory management. Rust’s type-system statically tracks object lifetimes and ownership [26], C programmers are responsible for manually managing memory, and though C++ provides memory-safe abstractions, C++ applications can also freely mix them with raw pointers. Crucially, when migrating C/C++ systems to Rust, developers need to reconcile these differences through the FFI layer, which is difficult to get right—for example, sharing pointers across FFI boundaries can lead to cross-language memory management issues, in which pointers allocated by one language are freed by the other [31]. Things get even more complicated when C and Rust code attempt to share ownership of memory.

Shared Ownership. rustTLS allows clients to create certificate verifiers and share them across server configurations. To enable this sharing, rustTLS represents these verifiers using atomic reference-counters (Arc), so that their memory is automatically reclaimed when they are no longer referenced.
<table>
<thead>
<tr>
<th>Safety Category</th>
<th>Bugs and Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception Safety</td>
<td>hyper-#2397, rustls-#886, rust-#74990, articchoke-#35.</td>
</tr>
<tr>
<td>Type Safety &amp; Invariants</td>
<td>Read-only data: rustls-ffi-#, encoding-c-L1009. String conventions: rustls-ffi-L169, Bugzilla-1374629. Type confusion: okcam-#1801.</td>
</tr>
</tbody>
</table>

Table 1: A taxonomy of memory-safety issues that occur in the interaction of C/C++ with Rust.

```rust
1 // (1) Create a verifier by cloning a certificate store.
2 pub extern "C" fn rustls_client_cert_verifier_new(
3     store: *const rustls_root_cert_store,
4 ) -> *const rustls_client_cert verifier {
5     let store: &RootCertStore = try_ref_from_ptr!(store);
6     return Arc::into_raw(... store.clone() ...) as *const _;
7 }
8 // (2) "Free" a verifier. C code must consider this pointer
9 // unusable after this call and must not call this function twice.
10 pub extern "C" fn rustls_client_cert_verifier_free(
11     verifier: *const rustls_client_cert_verifier) {
12     ffi_panic_boundary! { // We construct an Arc and drop it (count goes from 1 to 0)
13         if verifier.is_null() { return; }
14         unsafe { drop(Arc::from_raw(verifier)) }; // We drop the Arc from the FFI
15     }
16 }
17 }
18 // (3) Similar to example to the above but without Arc.
19 pub extern "C" fn rustls_root_cert_store_free(store: ...) {
20     ffi_panic_boundary! { // 'Free' a store
21         let store = try_box_from_ptr!(store);
22         drop(store)
23     }
24 }
```

Figure 1: Example of safety issues in rusTLS FFI functions. Exception safety: (1) can unwind across the FFI boundary if clone runs out of memory. Temporal safety: (2) and (3) can result in use-after-free and double-free errors due to incorrect function parameters or repeated function calls.

However, rusTLS exposes pointers to these objects through its FFI and thus requires clients to explicitly relinquish them through the function `rustls_client_cert_verifier_free` in Figure 1. This function `unsafe` reconstructs the Arc reference from a raw pointer and immediately drops it, thus decreasing the reference count. Importantly, this function expects the count to be 1 (which is the caller’s copy), so when used correctly, this function will also drop the object referenced by the pointer. The caller can however misuse the function—for example by freeing the same pointer twice or reusing a freed pointer—causing a miscount in the number of references, and introducing double- and use-after-free vulnerabilities in “safe” parts of rusTLS. Currently, rusTLS cannot detect a double-free: reading the count of a “freed” Arc reference triggers undefined behavior in the first place [rusTLS-#32]. Moreover, C implementations of TLS libraries may not necessarily rely on special APIs to release these objects (as well as the objects they refer to), and simply require clients to use the standard `free` function instead. Replacing such C implementations with rusTLS in a system can easily result in cross-language memory corruptions and introduce new memory vulnerabilities in the system.

### 2.2 Exception Safety

Rust handles unrecoverable errors (usually expressed with the `panic!` macro or any number of panicking function calls, such as `unwrap` or integer addition) by unwinding the stack and calling destructors along the way. Importantly, unwinding across the FFI boundary is considered undefined behavior.

Though this is currently debated within the Rust community [project-ffi-unwind], FFIs should explicitly handle panics to guarantee exception safety—and ideally communicate a failure to the caller. Rust does not provide any particular support with this task though, and so it is entirely up to developers to enforce safety in their code. rustTLS for instance wraps fallible top-level external functions through the macro `ffi_panic_boundary!` (e.g., Figure 1), which catches any unwinding panic and returns a default value to the caller. Since many basic operations in Rust can possibly panic, it is easy to miss necessary handlers. For an explicit bug, note that `rustls_client_cert_verifier_new` in Figure 1 is not exception safe, as cloning `RootCertStore` may trigger an unhandled out-of-memory panic that unwinds across the FFI.

### 2.3 Rust Invariants and Type Safety

Idiomatic Rust code heavily relies on the invariants guaranteed by the type system to ensure memory safety and correctness. Since C/C++ programs do not generally follow the same invariants [8], C/C++ may violate them when interacting with Rust code, especially after rewrites.

---

1The situation is even worse when handling recoverable errors, as there are no guidelines or tools to help with this. Discussed in detail in Section 4.
Aliasing. Function `decode_to_utf8` (Figure 2) decodes the content of an immutable slice `&[u8]` into a mutable slice `&mut [u8]`. Rust aliasing discipline ensures that these slices do not overlap which allows, among other things, numerous compiler optimizations. But these conditions are not checked or guaranteed by `decoder_decode_to_utf8` when reconstructing the slices through unsafe functions `from_raw_parts` and `from_raw_parts_mut`. The wrapper replaces buffer slices with C-compatible equivalent types—raw pointers and their respective lengths—which may alias or overlap. As with the example in Section 1, this can result in undefined behavior in the Rust FFI and unsafe optimizations by LLVM.

2.4 Other Undefined Behavior

Some sources of undefined behavior are more subtle and involve details of the languages involved and specific conventions of the architecture’s ABIs.

Glue code. A common source of issues in the examples discussed above is that glue code needs to use unsafe APIs to reconstruct Rust abstractions. Unsafe functions shift the responsibility of ensuring safety from the compiler to developers who design these interfaces independently from the applications that use them and thus incorporate must critical assumptions in the interface itself. However, most of these assumptions (e.g., pointers’ lifetime, ownership, and bounds), cannot be validated at runtime—Rust does not provide constructs to check them—and so FFI functions (implicitly) trust the caller and assume that inputs are valid. This trust is misplaced: FFIs represent a boundary between safe Rust components and arbitrary, untrusted code. Thus, caller code may pass invalid inputs that can easily break Rust’s safety guarantees, which defeats the main reason for migrating unsafe libraries to Rust and creates ideal conditions for cross-language attacks [33].

ABI Compatibility. ABI-level optimizations can be problematic in C/C++/Rust systems, in which components are compiled with different compilers and possibly incompatible optimizations. For example, on 64-bit architectures compilers can pack consecutive 32-bits function arguments into a single 64-bit register and thus reduce register pressure [9]. Cross-language function calls can however result in undefined behavior, if the respective compilers do not pack function inputs in the same way. For example, though C’s `size_t` are Rust `u32` types are both 32-bits, C compilers pack them but `rustc` does not [ockam-#1791].

3 \(R^3\) DESIGN

While one could imagine modifications to the FFI assumptions made by Rust to guard against individual errors discussed in Section 2, a solution entirely within Rust is likely impossible in general. This is because many of the FFI bugs surveyed are fundamentally cross-language issues. Instead, we propose that both C and Rust must conform to a shared, formally-based domain-specific language, which provides a safe-by-construction abstraction over the FFI boundary.

Concretely, \(R^3\)’s goal is to ensure that Rust code is not the source of any new memory safety issues in the application. To achieve this goal, \(R^3\) consists of:

1. An allocation tracker integrated into the C/C++ application’s memory allocator, which allows querying the application’s current allocations and their lengths.
2. A type system using refinement types [10, 25, 41] that ensures that developers writing FFI code perform adequate security checks to ensure safety, including establishing Rust’s memory safety invariants.

\(R^3\)’s allocation tracker. The allocation tracker is a runtime component that tracks allocations made by the C/C++ application code, so that their metadata can be queried by Rust FFI code, and to enforce cross-language temporal safety. This allocation tracker can be implemented simply by wrapping calls to `malloc`, `free`, and related functions in the C/C++ program, as done in numerous security frameworks [27, 34, 40]. For example, to prevent cross-language frees, the allocator keeps track of the set of pointers converted to Rust references (`currentRefs`) and then \(R^3\)’s wrapped `free` ensures that C/C++ code never frees allocations currently owned by Rust.

\(R^3\)’s refinement type system. \(R^3\) annotates Rust’s unsafe FFI functions with refinement types that ensure Rust unsafe features are used safely in FFI code. Refinement types such as Flux [30] are an extension to Rust type system that allows types to be decorated with logical predicates that constraint the set of values represented by a type. For example, we can use the refinement type `ptr: *mut T | not_null(ptr)` to represent the set of non-null pointers to type `T`. Importantly, to establish that an arbitrary pointer has such a refinement, the refinement type-system requires the code to explicitly check that condition. Thus, using refinement types...
we can ensure that before calling unsafe functions such as
std::slice::from_raw_parts_mut, the inputs are appropriately sanitized by calling the allocator tracker to check the relevant properties.

We show how $R^3$ can prevent various FFI safety issues by revisiting the example from Section 2.

**Temporal Safety.** To prevent temporal-safety vulnerabilities arising due to sharing memory ownership (Section 2.1), $R^3$ provides FFI-safe Arc implementations. Intuitively, safe APIs Arc::into_raw_safe and Arc::from_raw_safe allow FFI code to convert from standard Rust Arc references to C/C++ pointers and back, while keeping the reference count accurate. In addition to the reference count, these APIs keep track of the address of the references converted into raw pointers in the allocator state, and thus can ensure that only valid pointers can be converted back to Rust references. For example, using these APIs, the FFI code from Figure 1 can automatically eliminate the double-free vulnerability in function rustls_client_cert_verifier_free.

**Writing safe FFI in $R^3$.** To enforce safety in the FFI from Figure 2, we must ensure that the input pointers and lengths parameters designate valid memory regions and respect Rust’s non-aliasing expectations. $R^3$ statically ensures these conditions through the refinements in the type signature of from_raw_parts_mut_safe in Figure 3, which provides a safe interface to std::slice::from_raw_parts_mut. In a nutshell, those refinements disallow invalid inputs and require the FFI wrapper decoder_decode_to_utf8 to perform safety checks to validate them, as shown in Figure 3.

**Spatial safety.** For example, to establish the pre-condition not_null(data), $R^3$ requires the wrapper to perform a null-check on the pointer data—failing to do so results in a compile-time error. The more complicated invariant valid_cpp_alloc requires querying the allocator tracker to determine that (data,len) is contained within a single allocated object.

**Non-aliasing invariants.** In contrast to the local safety issues discussed above, some Rust invariants (e.g., non-aliasing references) require global, context-sensitive reasoning. For example, to correctly establish the pre-condition not_aliased, $R^3$ tracks all pointer and length pairs converted in the current context through the thread-local global variable current_refs. Automatic modifications to this variable are then captured via the post-condition predicate add_to_current_refs, which allows the code above to type-check.

**Exception safety.** $R^3$ forces all FFI Rust functions to catch and handle panics at the boundary so that these are not propagated to the C/C++ application. This is possible because $R^3$ requires all FFI Rust functions to be annotated (as shown in Figure 3) with r3_safeffi. Developers, however, are still responsible for suitably responding to the panic, for instance, by exiting the process or transforming this into an error code for the application.

4 DISCUSSION

There are a number of important challenges for securing the C-Rust FFI beyond what we have discussed here—and beyond what $R^3$ addresses. For example, while we have discussed handling panics, handing recoverable exceptions in FFI code is far more difficult and typically requires that the FFI code translate this exception into an actionable error to the host application. Existing solutions are ad-hoc and typically require duplicating error-handling code on both sides of the FFI, and are thus prone to bugs [ExternError]. Depending on the application, idiomatic solution could automatic translate Rust panics into corresponding C++ exceptions, or convert them into a suitable C error-codes. Another challenge is categorizing possible errors that could occur when C++/C applications expose callbacks to the Rust code, which we believe could introduce additional classes of safety errors.

As code written in Rust becomes more common, the interaction between other languages and Rust will only continue to become an attack surface. When manually writing Rust FFI code today, it is very easy to introduce (or re-introduce) memory safety bugs: we hope that future tools like $R^3$ will help developers write secure FFI code, and as a consequence, actually deliver on the promise and safety guarantees of Rust.

REFERENCES


