◂ Back to all articles

Published on 2024-05-03

C C++
Rust
Rewrite
Safety

How to rewrite a C++ codebase successfully

Table of contents

Discussions: /r/programming, /r/rust, Lobsters

Not your typical 'Rewrite it in Rust' article.

I recently wrote about inheriting a legacy C++ codebase. At some point, although I cannot pinpoint exactly when, a few things became clear to me:

- No one in the team but me is able - or feels confident enough - to make a change in this codebase
- This is a crucial project for the company and will live for years if not decades
- The code is pretty bad on all the criteria we care about: correctness, maintainability, security, you name it. I don't blame the original developers, they were understaffed and it was written as a prototype (the famous case of the prototype which becomes the production code).
- No hiring of C++ developers is planned or at least in the current budget (also because that's the only C++ project we have and we have many other projects to maintain and extend)

So it was apparent to me that sticking with C++ was a dead end. It's simply a conflict of values and priorities: C++ values many things that are not that important in this project, such as performance above all; and it does not give any guarantees about things that are crucial to us, such as memory and temporal safety (special mention to integer under/overflows. Have fun reviewing every single instance of arithmetic operations to check if it can under/overflow).

We bought a race car but what we needed was a family-friendly 5 seater, that's our mistake.

The only solution would be to train everyone in the team on C++ and dedicate a significant amount of time rewriting the most problematic parts of the codebase to perhaps reach a good enough state, and even then, we'd have little confidence our code is robust against nation-state attacks.

It's a judgment call in the end, but that seemed to be more effort than 'simply' introducing a new language and doing a rewrite.

I don't actually like the term 'rewrite'. Folks on the internet will eagerly repeat that rewrites are a bad idea, will undoubtedly fail, and are a sign of hubris and naivety. I have experienced such rewrites, from scratch, and yes that does not end well.

However, I claim, because I've done it, and many others before me, that an **incremental** rewrite can be successful, and is absolutely worth it. It's all about how it is being done, so here's how I proceeded and I hope it can be applied in other cases, and people find it useful.

I think it's a good case study because whilst not a big codebase, it is a complex codebase, and it's used in production on 10+ different operating systems and architectures, including by external customers. This is not a toy.

So join me on this journey, here's the guide to rewrite a C++ codebase successfully. And also what not do!

The project

This project is a library that exposes a C API but the implementation is C++, and it vendors C libraries (e.g. mbedtls) which we build from source. The final artifacts are a libfoo.a static library and a libfoo.h C header. It is used to talk to applets on a smart card like your credit card, ID, passport or driving license (yes, smart cards are nowadays everywhere - you probably carry several on you right now), since they use a ~~bizarre~~ interesting communication protocol. The library also implements a home-grown protocol on top of the well-specified smart card protocol, encryption, and business logic.

It is meant to be part of an user-facing application running on smartphones and Point of Sales terminals, as well as in servers running in a datacenter or in the cloud.

This library is used in:

- Android applications, through JNI
- Go back-end services running in Kubernetes, through CGO
- iOS applications, through Swift FFI
- C and C++ applications running on very small 32 bits ARM boards similar to the first Raspberry Pi

Additionally, developers are using macOS (x64 and arm64) and Linux so the library needs to build and run on these platforms.

Since external customers also integrate their applications with our library and we do not control these environments, and because some developer machines and servers use glibc and others musl, we also need to work with either the glibc and the musl C libraries, as well as clang and gcc, and expose a C89-ish API, to maximize compatibility.

Alright, now that the stage is set, let's go through the steps of rewriting this project.

Improve the existing codebase

That's basically all the steps in Inheriting a legacy C++ codebase. We need to start the rewrite with a codebase that builds and runs on every platform we support, with tests passing, and a clear README explaining how to setup the project locally. This is a small investment (a few days to a few weeks depending on the scale of the codebase) that will pay massive dividends in the future.

But I think the most important point is to trim all the unused code which is typically the majority of the codebase! No one wants to spend time and effort on rewriting completely unused code.

Additionally, if you fail to convince your team and the stakeholders to do the rewrite, you at least have improved the codebase you are now stuck with. So it's time well spent either way.

Get buy-in

Same as in my previous article: Buy-in from teammates and stakeholders is probably the most important thing to get, and maintain.

It's a big investment in time and thus money we are talking about, it can only work with everyone on board.

Here I think the way to go is showing the naked truth and staying very factual, in terms managers and non-technical people can understand. This is roughly what I presented:

- The bus factor for this project is 1 (me)
- Tool X shows that there are memory leaks at the rate of Y MiB/hour which means the application using our library will be OOM killed after around Z minutes/hours.
- Quick and dirty fuzzing manages to make the library crash 133 times in 10 seconds
- Linter X detects hundreds of real issues we need to fix
- All of these points make it really likely a hacker can exploit our library to gain Remote Code Execution (RCE) or steal secrets

Essentially, it's a matter of genuinely presenting the alternative of rewriting being cheaper in terms of time and effort compared to improving the project with pure C++. If your teammates and boss are reality-based, it should be a straightforward decision.

We use at my day job basically a RFC process to introduce a major change. That's great because it forces the person pushing for a change to document the current issues, the possible solutions, and allowing for a rational discussion to take place in the team. And documenting the whole process in a shared document (that allows comments) is very valuable because when people ask about it months later, you can just share the link to it.

After the problematic situation has been presented, I think at least 3 different solutions should be presented and compared (including sticking with pure C++), and seriously consider each option. I find it important here to be as little emotionally invested as possible even if one option is your favorite, and to be ready to work for possibly months on your least favorite option, if it happens to be chosen by the collective.

Ideally, if time permits, a small prototype for the preferred solution should be done, to confirm or infirm early that it can work, and to eliminate doubts. It's a much more compelling argument to say: "Of course it will work, here is prototype I made, let's look at it together!" compared to "I hope it will work, but who knows, oh well I guess we'll see 3 months in...".

After much debate, we settled on Rust as the new programming language being introduced into the codebase. It's important to note that I am not a Rust die hard fan. I appreciate the language but it's not perfect (see the FFI section later), it has issues, it's just that it solves all the issues we have in this project, especially considering the big focus on security (since we deal with payments), the relative similarity with the company tech stack (Go), and the willingness of the team to learn it and review code in it.

After all, the goal is also to gain additional developers, and stop being the only person who can even touch this code.

I also seriously considered Go, but after doing a prototype, I was doubtful the many limitations of CGO would allow us to achieve the rewrite. Other teammates also had concerns on how the performance and battery usage would look like on low-end Android and Linux devices, especially 32 bits, having essentially two garbage collectors running concurrently, the JVM one and the Go one.

Keeping buy-in

Keeping buy-in after initially getting it is not a given, since software always takes longer than expected and unexpected hurdles happen all the time. Here, showing the progress through regular demos (weekly or biweekly is a good frequency) is great for stakeholders especially non-technical ones. And it can potentially motivate fellow developers to also learn the new language and help you out.

Additionally, showing how long-standing issues in the old code get automatically solved by the new code, e.g. memory leaks, or fuzzing crashes in one function, are a great sign for stakeholders of the quality improving and the value of the on-going effort.

Be prepared to repeat many many times the decision process that led to the rewrite to your boss, your boss's boss, the odd product manager who's not technical, the salesperson supporting the external customers, etc. It's important to nail the elevator's pitch.

That applies also to teammates, who might be unsure the new programming language 'carries its weight'. It helps to regularly ask them how they feel about the language, the on-going-effort, the roadmap, etc. Also, pairing with them, so that ideally, everyone in the team feels confident working on this project alone.

Preparations to introduce the new language

Before adding the first line of code in the new language, I created a Git tag last-before-rust. The commit right after introduced some code in Rust.

This proved invaluable, because when rewriting the legacy code, I found tens of bugs lying around, and I think that's very typical. Also, this rewriting effort requires time, during which other team members or external customers may report bugs they just found.

Every time such a bug appeared, I switched to this Git tag, and tried to reproduce the bug. Almost every time, the bug was already present before the rewrite. That's a very important information (for me, it was a relief!) for solving the bug, and also for stakeholders. That's the difference in their eye between: We are improving the product by fixing long existing bugs; or: we are introducing new bugs with our risky changes and we should maybe stop the effort completely because it's harming the product.

Furthermore, I think the first commit introducing the new code should add dummy code and focus on making the build system and CI work seamlessly on every supported platform. This is not appealing work but it's necessary. Also, having instructions in the README explaining a bit what each tool does (cargo, rustup, clippy, etc) is very valuable and will ease beginners into contributing in the new language.

Incremental rewrite

Along with stakeholder buy-in, the most important point in the article is that only an **incremental** rewrite can succeed, in my opinion. Rewriting from scratch is bound to fail, I think. At least I have never seen it succeed, and have seen it fail many times.

What does it mean, very pragmatically? Well it's just a few rules of thumb:

- A small component is picked to be rewritten, the smallest, the better. Ideally it is as small as one function, or one class.
- The new implementation is written in the same Git (or whatever CVS you use) repository as the existing code, alongside it. It's a 'bug for bug' implementation which means it does the exact same thing as the old implementation, even if the old seems sometimes non-sensical. In some cases, what the old code tries to do is so broken and insecure, that we have to do something different in the new code, but that should be rare.
- Tests for the new implementation are written and pass (so that we know the new implementation is likely correct)
- Each site calling the function/class is switched to using the new implementation. After each replacement, the test suite is run and passes (so that we know that nothing broke at the scale of the project; a kind of regression testing). The change is committed. That way is something breaks, we know exactly which change is the culprit.
- A small PR is opened, reviewed and merged. Since our changes are anyways incremental, it's up to us to decide that the current diff is of the right size for a PR. We can make the PR as big or small as we want. We can even make a PR with only the new implementation that's not yet used at all.
- Once the old function/class is not used anymore by any code, it can be 'garbage-collected' i.e. safely removed. This can even be its own PR depending on the size.
- Rinse and repeat until all of the old code has been replaced

There are of course thornier cases, but that's the gist of it. What's crucial is that each commit on the main branch builds and runs fine. At not point the codebase is ever broken, does not build, or is in an unknown state.

It's actually not much different from the way I do a refactor in a codebase with just one programming language.

What's very important to avoid are big PRs that are thousands lines long and nobody wants to review them, or long running branches that effectively create a multiverse inside the codebase. It's the same as regular software development, really.

Here are a few additional tips I recommend doing:

- Starting from the leaves of the call graph is much easier than from the root. For example, if foo calls bar which calls baz, first rewriting baz then bar then foo is straightforward, but the reverse is usually not true.

- Thus, mapping out at the start from a high-level what are the existing components and which component calls out to which other component is invaluable in that regard, but also for establishing a rough roadmap for the rewrite, and reporting on the progress ("3 components have been rewritten, 2 left to do!").
- Port the code comments from the old code to the new code if they make sense and add value. In my experience, a few are knowledge gems and should be kept, and most are completely useless noise.
- If you can use automated tools (search and replace, or tools operating at the AST level) to change every call site to use the new implementation, it'll make your reviewers very happy, and save you hours and hours of debugging because of a copy-paste mistake
- Since Rust and C++ can basically only communicate through a C API (I am aware of experimental projects to make them talk directly but we did not use those - we ultimately want 100% Rust code exposing a C API, just like the old C++ did), it means that each Rust function must be accompanied by a corresponding C function signature, so that C++ can call it as a C function. I recommend automating this process with cbindgen. I have encountered some limitations with it but it's very useful, especially to keep the implementation (in Rust) and the API (in C) in sync, or if your teammates are not comfortable with C.
- Automate when you can, for example I added the cbindgen code generation step to CMake so that rebuilding the C++ project would automatically run cbindgen as well as cargo build for the right target in the right mode (debug or release) for the right platforms (--target=...). DevUX matters!
- When rewriting a function/class, port the tests for this function/class to the new implementation to avoid reducing the code coverage each time
- Make the old and the new test suites fast so that the iteration time is short
- When a divergence is detected (a difference in output or side effects between the old and the new implementation), observe with tests or within the debugger the output of the old implementation (that's where the initial Git tag comes handy, and working with small commits) in detail so that you can correct the new implementation. Some people even develop big test suites verifying that the output of the old and the new implementation are exactly the same.
- Since it's a bug-for-bug rewrite, *what* the new implementation does may seem weird or unnecessarily convoluted but shall be kept (at least as a first pass). However, *how* it does it in the new code should be up to the best software engineering standards, that means tests, fuzzing, documentation, etc.
- Thread lightly, what can tank the project is being too bold when rewriting code and by doing so, introducing bugs or subtly changing the

behavior which will cause breakage down the line. It's better to be conservative here.

- Pick a prefix for all structs and functions in the C API exposed by the Rust code, even if it's just RUST_xxx, so that they are immediately identifiable and greppable. Just like libcurl has the prefix curl_xxx.

Finally, there is one hidden advantage of doing an incremental rewrite. A from-scratch rewrite is all or nothing, if it does not fully complete and replace the old implementation, it's useless and wasteful. However, an incremental rewrite is immediately useful, may be paused and continued a number of times, and even if the funding gets cut short and it never fully completes, it's still a clear improvement over the starting point.

Fuzzing

I am a fan a fuzzing, it's great. Almost every time I fuzz some code, I find a corner case I did not think about, especially when doing parsing.

I added fuzzing to the project so that every new Rust function is fuzzed. I initially used AFL but then turned to cargo-fuzz, and I'll explain why.

Fuzzing is only useful if code coverage is high. The worst that can happen is to dedicate serious time to setup fuzzing, to only discover at the end that the same few branches are always taken during fuzzing.

Coverage can only be improved if developers can easily see exactly which branches are being executed during fuzzing. And I could not find an easy way with AFL to get a hold on that data.

Using cargo-fuzz and various LLVM tools, I wrote a small shell script to visualize exactly which branches are taken during fuzzing as well as the code coverage in percents for each file and for the project as a whole (right now it's at around 90%).

To get to a high coverage, the quality of the corpus data is paramount, since fuzzing works by doing small mutations of this corpus and observing which branches are taken as a result.

I realized that the existing tests in C++ had lots of useful data in them, e.g.:

C++

```
    assert(foo(input) == ...);
```

So I had the idea of extracting all the input = ... data from the tests to build a good fuzzing corpus. My first go at it was a hand-written quick and dirty C++ lexer in Rust. It worked but it was clunky. Right after I finished it, I thought: why don't I use tree-sitter to properly parse C++ in Rust?

And so I did, and it turned out great, just 300 lines of Rust walking through each TestXXX.cpp file in the repository and using tree-sitter to extract each pattern. I used the query language of tree-sitter to do so:

Rust

```rust
let query = tree_sitter::Query::new(
    tree_sitter_cpp::language(),
    "(initializer_list (number_literal)+) @capture",
)
```

The tree-sitter website thankfully has a playground where I could experiment and tweak the query and see the results live.

As time went on and more and more C++ tests were migrated to Rust tests, it was very easy to extend this small Rust program that builds the corpus data, to also scan the Rust tests!

A typical Rust test would look like this:

Rust

```rust
const INPUT: [u8; 4] = [0x01, 0x02, 0x03, 0x04]; // <= This is the interesting data.
assert_eq!(foo(&INPUT), ...);
```

And the query to extract the interesting data would be:

Rust

```rust
let query = tree_sitter::Query::new(
    tree_sitter_rust::language(),
    // TODO: Maybe make this query more specific with:
    // `(let_declaration value: (array_expression (integer_literal)+)) @capture`.
    // But in a few cases, the byte array is defined with `const`, not `let`.
    "(array_expression (integer_literal)+) @capture",
)
```

However I discovered that not all data was successfully extracted. What about this code:

Rust

```rust
const BAR : u8 = 0x42;
const INPUT: [u8; 4] = [BAR, 0x02, 0x03, 0x04]; // <= This is the interesting data.
assert_eq!(foo(&INPUT), ...);
```

We have a constant BAR which trips up tree-sitter, because it only sees a literal (i.e. 3 letters: 'B', 'A' and 'R') and does not know its value.

The way I solved this issue was to do two passes: once to collect all constants along with their values in a map, and then a second pass to find all arrays in tests:

Rust

```rust
let query = tree_sitter::Query::new(
    tree_sitter_rust::language(),
    "(const_item value: (integer_literal)) @capture ",
)
```

So that we can then resolve the literals to their numeric value.

That's how I implemented a compiler for the Kotlin programming language in the past and it worked great. Maybe there are more advanced approaches but this one is dead-simple and fast so it's good enough for us.

I am pretty happy with how this turned out, scanning all C++ and Rust files to find interesting test data in them to build the corpus. I think this was key to move from the initial 20% code coverage with fuzzing (using a few hard-coded corpus files) to 90%. It's fast too.

Also, it means the corpus gets better each time we had a test (be it in C++ or Rust), for free.

Does it mean that the corpus will grow to an extreme size? Well, worry not, because LLVM comes with a fuzzing corpus minimizer:

Shell

```
place).
$ cargo +nightly fuzz cmin [...]
```

For each file in the corpus, it feeds it as input to our code, observes which branches are taken, and if a new set of branches is taken, this file remains (or perhaps gets minimized even more, not sure how smart this tool is). Otherwise it is deemed a duplicate and is trimmed.

So:

1. We generate the corpus with our program
2. Minimize it
3. Run the fuzzing for however long we wish. It runs in CI for every commit and developers can also run it locally.
4. When fuzzing is complete, we print the code coverage statistics

Finally, we still have the option to add manually crafted files to this corpus if we wish. For example after hitting a bug in the wild, and fixing it, we can add a reproducer file to the corpus as a kind of regression test.

Pure Rust vs interop (FFI)

Writing Rust has been a joy, even for more junior developers in the team. Pure Rust code was pretty much 100% correct on the first try.

However we had to use unsafe {} blocks in the FFI layer. We segregated all the FFI code to one file, and converted the C FFI structs to Rust idiomatic structs as soon as possible, so that the bulk of the Rust code can be idiomatic and safe.

But that means this FFI code is the most likely part of the Rust code to have bugs. To get some confidence in its correctness, we write Rust tests using the C FFI functions (as if we were a C consumer of the library) running under Miri which acts as valgrind essentially, simulating a CPU and checking that our code is memory safe. Tests run perhaps 5 to 10 times as slow as without Miri but this has proven invaluable since it detected many bugs ranging from alignment issues to memory leaks and use-after-free issues.

We run tests under Miri in CI to make sure each commit is reasonably safe.

So beware: introducing Rust to a C or C++ codebase may actually introduce new memory safety issues, usually all located in the FFI code.

Thankfully that's a better situation to be in than to have to inspect all of the codebase when a memory issue is detected.

C FFI in Rust is cumbersome and error-prone

The root cause for all these issues is that the C API that C++ and Rust use to call each other is very limited in its expressiveness w.r.t ownership, as well as many Rust types not being marked #[repr(C)], even types you would expect to, such as Option, Vec or &[u8]. That means that you have to define your own equivalent types:

Rust

```rust
#[repr(C)]
// An option type that can be used from C
pub struct OptionC<T> {
    pub has_value: bool,
    pub value: T,
}


#[repr(C)]
// Akin to `&[u8]`, for C.
pub struct ByteSliceView {
    pub ptr: *const u8,
    pub len: usize,
}

/// Owning Array i.e. `Vec<T>` in Rust or `std::vector<T>` in C++.
#[repr(C)]
pub struct OwningArrayC<T> {
    pub data: *mut T,
    pub len: usize,
    pub cap: usize,
}

/// # Safety
/// Only call from C.
#[no_mangle]
pub extern "C" fn make_owning_array_u8(len: usize) -> OwningArrayC<u8> {
    vec![0; len].into()
}
```

Apparently, Rust developers do not want to commit to a particular ABI for these types, to avoid missing out on some future optimizations. So it means that every Rust struct now needs the equivalent "FFI friendly" struct along with conversion functions (usually implemented as .into() for convenience):

Rust

```rust
struct Foo<'a> {
    x: Option<usize>,
    y: &'a [u8],
    z: Vec<u8>,
}


#[repr(C)]
struct FooC {
    x: OptionC<usize>,
    y: ByteSliceView,
    z: OwningArrayC<u8>,
}
```

Which is cumbersome but still fine, especially since Rust has powerful macros (which I investigated using but did not eventually). However, since Rust also does not have great idiomatic support for custom allocators, we stuck with the standard memory allocator, which meant that each struct with heap-allocated fields has to have a deallocation function:

Rust

```rust
#[no_mangle]
pub extern "C" fn foo_free(foo: &FooC) {
    ...
}
```

And the C or C++ calling code would have to do:

C++

```cpp
FooC foo{};
if (foo_parse(&foo, bytes) == SUCCESS) {
    // do something with foo...
    ...

    foo_free(foo);
}
```

To simplify this, I introduced a defer construct to C++ (thanks Gingerbill!):

C++

```cpp
FooC foo{};
defer({foo_free(foo);});

if (foo_parse(&foo, bytes) == SUCCESS) {
    // do something with foo...
    ...
}
```

Which feels right at home for Go developers, and is an improvement over the style in use in the old C++ code where it was fully manual calls to new/delete.

Still, it's more work than what you'd have to do in pure idiomatic Rust or C++ code (or even C code with arenas for that matter).

In Zig or Odin, I would probably have used arenas to avoid that, or a general allocator with defer.

An example of a real bug at the FFI boundary

More perniciously, it's easy to introduce memory unsafety at the FFI boundary. Here is a real bug I introduced, can you spot it? I elided all the error handling to make it easier to spot:

Rust

```rust
#[repr(C)]
struct BarC {
    x: ByteSliceView,
}

```

```rust
#[no_mangle]
unsafe extern "C" fn bar_parse(input: *const u8, input_len: usize, bar_c: &mut BarC) {
    let input: &[u8] = unsafe { std::slice::from_raw_parts(input, input_len) };

    let bar: Bar = Bar {
        x: [input[0], input[1]],
    };

    *bar_c = BarC {
        x: ByteSliceView {
            ptr: bar.x.as_ptr(),
            len: bar.x.len(),
        },
    };
}
```

clippy did not notice anything. address-sanitizer did not notice anything. However, both miri and valgrind did, and fuzzing crashed (which was not easy to troubleshoot but at least pinpointed to a problem).

So...found it? Still nothing? Well, let's be good developers and add a test for it:

Rust

```rust
#[test]
fn bar() {
    // This mimicks how C/C++ code would call our function.
    let mut bar_c = MaybeUninit::<BarC>::uninit();
    let input = [0, 1, 2];
    unsafe {
        bar_parse(
            input.as_ptr(),
            input.len(),
            bar_c.as_mut_ptr().as_mut().unwrap(),
        );
    }
```

```rust

        let bar_c = unsafe { bar_c.assume_init_ref() };
        let x: &[u8] = (&bar_c.x).into();
        assert_eq!(x, [0, 1].as_slice());
    }
```

If you're lucky, cargo test would fail at the last assertion saying that the value is not what we expected, but in my case it passed every time, and so the bug stayed undetected for a while. That's because we unknowingly introduced undefined behavior, and as such, how or if it manifests is impossible to tell.

Let's run the test with Miri:

Text

```
running 1 test
test api::tests::bar ... error: Undefined Behavior: out-of-bounds pointer use: alloc195648 has been freed, so this pointer is dangling
    --> src/tlv.rs:321:18
     |
321  |         unsafe { &*core::ptr::slice_from_raw_parts(item.ptr, item.len) }
     |                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ out-of-bounds pointer use: alloc195648 has been freed, so this pointer is dangling
     |
     = help: this indicates a bug in the program: it performed an invalid operation, and caused Undefined Behavior
     = help: see https://doc.rust-lang.org/nightly/reference/behavior-considered-undefined.html for further information
help: alloc195648 was allocated here:
    --> src/api.rs:1396:9
     |
1396 |     let bar: Bar = Bar {
     |         ^^^
help: alloc195648 was deallocated here:
    --> src/api.rs:1406:1
     |
1406 | }
```

Miri is great, I tell you.

The issue here is that we essentially return a pointer to local variable (x) from inside the function, so the pointer is dangling.

Alternatively we can call our function from C/C++ and run that under valgrind:

C

```c
int main() {
  BarC bar{};
  const uint8_t input[] = {0, 1, 2, 3};
  bar_parse(input, sizeof(input), &bar);
  assert(bar.x.ptr[0] == 0);
  assert(bar.x.ptr[1] == 1);
}
```

And I get:

Text

```
==805913== Conditional jump or move depends on uninitialised value(s)
==805913==    at 0x127C34: main (src/example.cpp:13)
==805913==
==805913== Conditional jump or move depends on uninitialised value(s)
==805913==    at 0x127C69: main (src/example.cpp:14)
```

Which is not very informative, but better than nothing. Miri's output is much more actionable.

So in conclusion, Rust's FFI capabilities work but are tedious are error-prone in my opinion, and so require extra care and testing with Miri/fuzzing, with high code coverage of the FFI functions. It's not enough to only test the pure (non FFI) Rust code.

Another example of a real bug at the FFI boundary

When I started this rewrite, I was under the impression that the Rust standard library uses the C memory allocator (basically, malloc) under the covers when it needs to allocate some memory.

However, I quickly discovered that it is not (anymore?) the case, Rust uses its own allocator - at least on Linux where there is no C library shipping with the kernel. Miri again is the MVP here since it detected the issue of mixing the C and Rust allocations which prompted this section.

As Bryan Cantrill once said: "glibc on Linux, it's just, like, your opinion dude". Meaning, glibc is just one option, among many, since Linux is just the kernel and does not ship with a libC. So the Rust standard library cannot expect a given C library on every Linux system, like it would be on macOS or the BSDs or Illumos. All of that to say: Rust implements its own memory allocator.

The consequence of this, is that allocating memory on the C/C++ side, and freeing it on the Rust side, is undefined behavior: it amounts to freeing a pointer that was never allocated by this allocator. And vice-versa, allocating a pointer from Rust and freeing it from C.

That has dire consequences since most memory allocators do not detect this in release mode. You might free completely unrelated memory leading to use-after-free later, or corrupt the memory allocator structures. It's bad.

Here's a simplified example of code that triggered this issue:

Rust

```rust
#[repr(C)]
pub struct FooC {
    foo: u8,
    bar: *mut usize,
}

#[no_mangle]
pub extern "C" fn parse_foo(in_bytes: *const u8, in_bytes_len: usize, foo: &mut FooC) {
    let in_bytes: &[u8] = unsafe { &*core::ptr::slice_from_raw_parts(in_bytes, in_bytes_len) };

    // Parse `foo` from `in_bytes` but `bar` is sometimes not present in the payload.
    // In that case it is set manually by the calling code.
    *foo = FooC {
        foo: in_bytes[0],
        bar: if in_bytes_len == 1 {
            core::ptr::null_mut()
        } else {
            let x = Box::new(in_bytes[1] as usize);
            Box::into_raw(x)
        },
    }
}

#[no_mangle]
pub extern "C" fn free_foo(foo: &mut FooC) {
    if !foo.bar.is_null() {
        unsafe {
            let _ = Box::from_raw(foo.bar);
        }
    }
}
```

And the calling code:

C++

```
FooC foo{};

const uint8_t data[] = { 1 };
parse_foo(data, sizeof(data), &foo);
if (foo.bar == nullptr) {
  foo.bar = new size_t{99999};
}

free_foo(&foo);
```

This is undefined behavior if the array is of size 1, since in that case the Rust allocator will free a pointer allocated by the C allocator, and address sanitizer catches it:

Text

```
SUMMARY: AddressSanitizer: alloc-dealloc-mismatch /home/runner/work/llvm-project/llvm-project/final/llvm-project/compiler-rt/lib/asan/asan_malloc_linux.cpp:52:3 in free
```

However, it is only detected with sanitizers on and if a test (or fuzzing) triggers this case. Or by Miri if a Rust test covers this function.

So I recommend sticking to one 'side', be it C/C++ or Rust, of the FFI boundary, to allocate and free all the memory used in FFI structures. Rust has an edge here since the long-term goal is to have 100% of Rust so it will have to allocate all the memory anyway in the end.

Depending on the existing code style, it might be hard to ensure that the C/C++ allocator is not used at all for structures used in FFI, due to abstractions and hidden memory allocations.

One possible solution (which I did not implement but considered) is making FFI structures a simple opaque pointer (or 'handle') so that the caller has to use FFI functions to allocate and free this structure. That also means implementing getter/setters for certain fields since the structures are now opaque. It maximizes the ABI compatibility, since the caller cannot rely on a given struct size, alignment, or fields.

However that entails more work and more functions in the API.

libcurl is an example of such an approach, libuv is an example of a library which did not do this initially, but plans to move to this approach in future versions, which would be a breaking change for clients.

So to summarize, Miri is so essential that I don't know whether it's viable to write Rust code with lots of FFI (and thus lots of unsafe blocks) without it. If Miri did not exist, I would seriously consider using only arenas or reconsider the use of Rust.

Cross-compilation

Rust has great cross-compilation support; C++ not so much. Nonetheless I managed to coerced CMake into cross-compiling to every platform we support from my Linux laptop. After using Docker for more than 10 years I am firmly against using Docker for that, it's just clunky and slow and not a good fit. Also we already have to cross-compile to the mobile platforms anyway so why not make that work for all platforms?

That way, I can even cross-compile tests and example programs in C or C++ using the library and run them inside qemu to make sure all platforms work as expected.

I took inspiration from the CMake code in the Android project, which has to cross-compile for many architectures. Did you know that Android supports x86 (which is 32 bits), x86_64, arm (which is 32 bits), aarch64 (sometimes called arm64), and more?

In short, you instruct CMake to cross-compile by supplying on the command-line the variables CMAKE_SYSTEM_PROCESSOR and CMAKE_SYSTEM_NAME, which are the equivalent of GOARCH and GOOS if you are familiar with Go. E.g.:

Shell

```shell
$ cmake -B .build -S src -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_SYSTEM_NAME=Linux -DCMAKE_SYSTEM_PROCESSOR=arm
```

On the Rust side, you tell cargo to cross-compile by supplying the --target command-line argument, e.g.: --target=x86_64-unknown-linux-musl. This works by virtue of installing the pre-compiled toolchain for this platform with rustup first:

Shell

```shell
$ rustup target add x86_64-unknown-linux-musl
```

So now we have to convert in CMake CMAKE_SYSTEM_ARCHITECTURE and CMAKE_SYSTEM_NAME into a target triple that clang and cargo can understand. Of course you have to do all the hard work yourself. This is complicated by lots of factors like Apple using the architecture name arm64 instead of aarch64, iOS peculiarities, soft vs hard float, arm having multiple variants (v6, v7, v8, etc), and so on. Your mileage may vary. We opt-in into using musl with a CMake command line option, on Linux.

Here it is in all its glory:

Cmake

```cmake
# We need to craft the target triple to make it work when cross-compiling.
# NOTE: If an architecture supports both soft-float and hard-float, we pick hard-float (`hf`).
# since we do not target any real hardware with soft-float.
# Linux has two main libcs, glibc (the default) and musl (opt-in with `FMW_LIBC_MUSL=1`), useful for Alpine.
if (CMAKE_SYSTEM_NAME STREQUAL "Linux" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "x86_64" AND NOT DEFINED FMW_LIBC_MUSL)
    set(TARGET_TRIPLE "x86_64-unknown-linux-gnu")
elseif (CMAKE_SYSTEM_NAME STREQUAL "Linux" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "x86_64" AND "${FMW_LIBC_MUSL}" EQUAL 1)
```

```
        set(TARGET_TRIPLE "x86_64-unknown-linux-musl")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Linux" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "arm" AND NOT DEFINED FMW_LIBC_MUSL)
        set(TARGET_TRIPLE "arm-unknown-linux-gnueabihf")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Linux" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "arm" AND "${FMW_LIBC_MUSL}" EQUAL 1)
        set(TARGET_TRIPLE "arm-unknown-linux-musleabihf")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Linux" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "aarch64" AND NOT DEFINED FMW_LIBC_MUSL)
        set(TARGET_TRIPLE "aarch64-unknown-linux-gnu")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Linux" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "aarch64" AND "${FMW_LIBC_MUSL}" EQUAL 1)
        set(TARGET_TRIPLE "aarch64-unknown-linux-musl")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Linux" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "armv7")
        set(TARGET_TRIPLE "armv7-unknown-linux-gnueabihf")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Darwin" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "aarch64")
        set(TARGET_TRIPLE "aarch64-apple-darwin")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Darwin" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "arm64")
        set(TARGET_TRIPLE "aarch64-apple-darwin")
    elseif (CMAKE_SYSTEM_NAME
STREQUAL "Darwin" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "x86_64")
        set(TARGET_TRIPLE "x86_64-apple-darwin")
    elseif (CMAKE_SYSTEM_NAME STREQUAL "iOS" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "x86_64")
        set(TARGET_TRIPLE "x86_64-apple-ios")
        execute_process(COMMAND xcrun --sdk iphonesimulator --show-sdk-path OUTPUT_VARIABLE CMAKE_OSX_SYSROOT)
        string(REPLACE "\n" "" CMAKE_OSX_SYSROOT ${CMAKE_OSX_SYSROOT})
    elseif (CMAKE_SYSTEM_NAME STREQUAL "iOS" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "aarch64")
        set(TARGET_TRIPLE "aarch64-apple-ios")
        execute_process(COMMAND xcrun --sdk iphoneos --show-sdk-path OUTPUT_VARIABLE CMAKE_OSX_SYSROOT)
        string(REPLACE "\n" "" CMAKE_OSX_SYSROOT ${CMAKE_OSX_SYSROOT})
    elseif (CMAKE_SYSTEM_NAME STREQUAL "iOS" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "arm64")
        set(TARGET_TRIPLE "aarch64-apple-ios")
        execute_process(COMMAND xcrun --sdk iphoneos --show-sdk-path OUTPUT_VARIABLE CMAKE_OSX_SYSROOT)
        string(REPLACE "\n" "" CMAKE_OSX_SYSROOT ${CMAKE_OSX_SYSROOT})
    elseif (CMAKE_SYSTEM_NAME STREQUAL "Android" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "arm")
        set(TARGET_TRIPLE "arm-linux-androideabi")
```

```
elseif (CMAKE_SYSTEM_NAME STREQUAL "Android" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "armv7")
    set(TARGET_TRIPLE "armv7-linux-androideabi")
elseif (CMAKE_SYSTEM_NAME STREQUAL "Android" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "armv7-a")
    set(TARGET_TRIPLE "armv7-linux-androideabi")
elseif (CMAKE_SYSTEM_NAME STREQUAL "Android" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "aarch64")
    set(TARGET_TRIPLE "aarch64-linux-android")
elseif (CMAKE_SYSTEM_NAME STREQUAL "Android" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "i686")
    set(TARGET_TRIPLE "i686-linux-android")
elseif (CMAKE_SYSTEM_NAME STREQUAL "Android" AND CMAKE_SYSTEM_PROCESSOR STREQUAL "x86_64")
    set(TARGET_TRIPLE "x86_64-linux-android")
else()
    message(FATAL_ERROR "Invalid OS/Architecture, not supported: CMAKE_SYSTEM_NAME=${CMAKE_SYSTEM_NAME} CMAKE_SYSTEM_PROCESSOR=${CMAKE_SYSTEM_PROCESSOR}")
endif()

message(STATUS "Target triple: ${TARGET_TRIPLE}")

# If we are cross compiling manually (e.g to Linux arm), `CMAKE_C_COMPILER_TARGET` and `CMAKE_CXX_COMPILER_TARGET` are unset and we need to set them manually.
```

```
  # But if we are cross compiling through a separate build system e.g. to Android or iOS, they will set these variables and we should not override them.
  if ( NOT DEFINED CMAKE_C_COMPILER_TARGET )
      set(CMAKE_C_COMPILER_TARGET ${TARGET_TRIPLE})
  endif()
  if ( NOT DEFINED CMAKE_CXX_COMPILER_TARGET )
      set(CMAKE_CXX_COMPILER_TARGET ${TARGET_TRIPLE})
  endif()
```

There was a lot of trial and error as you can guess.

Also, gcc is not directly supported for cross-compilation in this approach because gcc does not support a --target option like clang does, since it's not a cross-compiler. You have to download the variant you need e.g. gcc-9-i686-linux-gnu to compile for x86, and set CMAKE_C_COMPILER and CMAKE_CXX_COMPILER to gcc-9-i686-linux-gnu. However, in that case you are not setting CMAKE_SYSTEM_NAME and CMAKE_SYSTEM_PROCESSOR since it's in theory not cross-compiling, so cargo will not have its --target option filled, so it won't work for the Rust code. I advise sticking with clang in this setup. Still, when not cross-compiling, gcc works fine.

Finally, I wrote a Lua script to cross-compile for every platform we support to make sure I did not break anything. I resorted to using the Zig toolchain (not the language) to be able to statically link with musl or cross-compile from Linux to iOS which I could not achieve with pure clang. However this is only my local setup, we do not use the Zig toolchain when building the production artifacts (e.g. the iOS build is done in a macOS virtual machine, not from a Linux machine).

This is very useful also if you have several compile-time feature flags and want to build in different configurations for all platforms, e.g. enable/disable logs at compile time:

Lua

```lua
local android_sdk = arg[1]
if android_sdk == nil or android_sdk == "" then
  print("Missing Android SDK as argv[1] e.g. '~/Android/Sdk/ndk/21.4.7075529'.")
  os.exit(1)
end

local build_root = arg[2]
if build_root == nil then
  build_root = "/tmp/"
end

local rustup_targets = {
  "aarch64-apple-darwin",
  "aarch64-linux-android",
  "aarch64-unknown-linux-gnu",
  "aarch64-unknown-linux-musl",
  "arm-linux-androideabi",
  "arm-unknown-linux-gnueabihf",
  "arm-unknown-linux-musleabihf",
  "armv7-linux-androideabi",
  "armv7-unknown-linux-gnueabi",
  "armv7-unknown-linux-gnueabihf",
  "armv7-unknown-linux-musleabi",
  "armv7-unknown-linux-musleabihf",
  "i686-linux-android",
  "x86_64-apple-darwin",
  "x86_64-linux-android",
  "x86_64-unknown-linux-gnu",
  "x86_64-unknown-linux-musl",
}

for i = 1,#rustup_targets do
  local target = rustup_targets[i]
  os.execute("rustup target install " .. target)
end

```

```lua

local targets = {
    {os="Linux", arch="x86_64", cc="clang", cxx="clang++", cmakeArgs=""},
    {os="Linux", arch="aarch64", cc="clang", cxx="clang++", cmakeArgs=""},
    {os="Linux", arch="arm", cc="clang", cxx="clang++", cmakeArgs=""},
    {os="Linux", arch="armv7", cc="clang", cxx="clang++", cmakeArgs=""},
    {os="Linux", arch="arm", cc="zig", cxx="zig", cmakeArgs="-DCMAKE_C_COMPILER_ARG1=cc -DCMAKE_CXX_COMPILER_ARG1=c++ -DFMW_LIBC_MUSL=1 -DCMAKE_C_COMPILER_TARGET=arm-linux-musleabihf -DCMAKE_CXX_COMPILER_TARGET=arm-linux-musleabihf"},
    {os="Linux", arch="aarch64", cc="zig", cxx="zig", cmakeArgs="-DCMAKE_C_COMPILER_ARG1=cc -DCMAKE_CXX_COMPILER_ARG1=c++ -DFMW_LIBC_MUSL=1 -DCMAKE_C_COMPILER_TARGET=aarch64-linux-musl -DCMAKE_CXX_COMPILER_TARGET=aarch64-linux-musl"},
    {os="Linux", arch="x86_64", cc="zig", cxx="zig", cmakeArgs="-DCMAKE_C_COMPILER_ARG1=cc -DCMAKE_CXX_COMPILER_ARG1=c++ -DFMW_LIBC_MUSL=1 -DCMAKE_C_COMPILER_TARGET=x86_64-linux-musl -DCMAKE_CXX_COMPILER_TARGET=x86_64-linux-musl"},
    {os="Darwin", arch="x86_64", cc="zig", cxx="zig", cmakeArgs="-DCMAKE_C_COMPILER_ARG1=cc -DCMAKE_CXX_COMPILER_ARG1=c++ -DFMW_LIBC_MUSL=1 -DCMAKE_C_COMPILER_TARGET=x86_64-macos-none -DCMAKE_CXX_COMPILER_TARGET=x86_64-
macos-none"},
    {os="Darwin", arch="arm64", cc="zig", cxx="zig", cmakeArgs="-DCMAKE_C_COMPILER_ARG1=cc -DCMAKE_CXX_COMPILER_ARG1=c++ -DFMW_LIBC_MUSL=1 -DCMAKE_C_COMPILER_TARGET=aarch64-macos-none -DCMAKE_CXX_COMPILER_TARGET=aarch64-macos-none"},
    {os="Android", arch="armv7-a", cc="clang", cxx="clang++", cmakeArgs="-DCMAKE_ANDROID_NDK='" .. android_sdk .. "'"},
    {os="Android", arch="aarch64", cc="clang", cxx="clang++", cmakeArgs="-DCMAKE_ANDROID_NDK='" .. android_sdk .. "'"},
    {os="Android", arch="i686", cc="clang", cxx="clang++", cmakeArgs="-DCMAKE_ANDROID_NDK='" .. android_sdk .. "'"},
    {os="Android", arch="x86_64", cc="clang", cxx="clang++", cmakeArgs="-DCMAKE_ANDROID_NDK='" .. android_sdk .. "'"},
  }

  for i = 1,#targets do
    local target = targets[i]
    local build_dir = ".build-" .. target.os .. "-" .. target.arch .. "-" .. target.cc .. "-" .. target.cxx .. "-" .. target.cmakeArgs
    build_dir = string.gsub(build_dir, "%s+", "_")
    build_dir = string.gsub(build_dir, "^./+", "_")
    build_dir = build_root .. "/" .. build_dir
    print(build_dir)

    local cmd_handle = io.popen("command -v llvm-ar")
    local llvm_ar = cmd_handle:read('*a')
    cmd_handle:close()
    llvm_ar = string.gsub(llvm_ar, "%s+$", "")

    local cmd_handle = io.popen("command -v llvm-ranlib")
    local llvm_ranlib = cmd_handle:read('*a')
    cmd_handle:close()
    llvm_ranlib = string.gsub(llvm_ranlib, "%s+$", "")

    local build_cmd = "cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -B '" .. build_dir .. "' -DCMAKE_AR=" .. llvm_ar .. " -DCMAKE_RANLIB=" .. llvm_ranlib .. " -DCMAKE_SYSTEM_NAME=" .. target.os .. " -DCMAKE_SYSTEM_PROCESSOR=" .. target.arch .. " -DCMAKE_C_COMPILER=" .. target.cc .. " -DCMAKE_CXX_COMPILER=" .. target.cxx .. " " ..  target.cmakeArgs .. " -S src/. -G Ninja"
    print(build_cmd)
    os.execute(build_cmd)

    -- Work-around for getting rid of mbedtls linker flags specific to Apple's LLVM fork that are actually not needed.
    if target.os == "Darwin" then
      os.execute("sed -i '" .. build_dir .. "/CMakeFiles/rules.ninja' -e 's/ -no_warning_for_no_symbols -c//g'")
    end

    os.execute("ninja -C '" .. build_dir .. "'")
  end
```

I look forward to only having Rust code and deleting all of this convoluted stuff.

That's something that people do not mention often when saying that modern C++ is good enough and secure enough. Well, first I disagree with this statement, but more broadly, the C++ toolchain to cross-compile sucks. You only have clang that can cross-compile in theory but in practice you have to resort to the Zig toolchain to automate cross-compiling the standard library etc.

Also, developers not deeply familiar with either C or C++ do not want to touch all this CMake/Autotools with a ten-foot pole. And I understand them. Stockholm syndrome notwithstanding, these are pretty slow, convoluted, niche programming languages and no one wants to actively learn and use them unless they have to.

Once you are used to simply typing go build or cargo build, you really start to ask yourself if those weird things are worth anyone's time.

Conclusion

The rewrite is not yet fully done, but we have already more Rust code than C++ code, and it's moving along nicely, at our own pace (it's not by far the only project we have on our lap). Once all C++ code is removed, we will do a final pass to remove the CMake stuff and build directly via cargo. We'll see if that works when integrating with other build systems e.g. Bazel for Android or Xcode for iOS.

Developers who learned Rust are overall very happy with it and did not have too many fights with the borrow checker, with one notable exception of trying to migrate a C struct that used an intrusive linked list (ah, the dreaded linked list v. borrow checker!). My suggestion was to simply use a Vec in Rust since the linked list was not really justified here, and the problem was solved.

Adding unit tests was trivial in Rust compared to C++ and as a result people would write a lot more of them. Built-in support for tests is expected in 2024 by developers. I don't think one single C++ test was written during this rewrite, now that I think of it.

Everyone was really satisfied with the tooling, even though having to first do rustup target add ... before cross-compiling tripped up a few people, since in Go that's done automatically behind the scenes (I think one difference is that Go compiles everything from source and so does not need to download pre-compiled blobs?).

Everyone also had an easy time with their text editor/IDE, Rust is ubiquitous enough now that every editor will have support for it.

All the tooling we needed to scan dependencies for vulnerabilities, linting, etc was present and polished. Shootout to osv-scanner from Google, which allowed us to scan both the Rust and C++ dependencies in the same project (and it evens supports Go).

As expected, developers migrating C++ code to Rust code had a breeze with the Rust code and almost every time asked for assistance when dealing with the C++ code. C++ is just too complex a language for most developers, especially compared to its alternatives.

CMake/Make/Ninja proved surprisingly difficult for developers not accustomed to them, but I mentioned that already. I think half of my time during this rewrite was actually spent coercing all the various build systems (Bazel/Xcode/CMake/cargo/Go) on the various platforms into working well together. If there is no one in the team who's really familiar with build systems, I think this is going to be a real challenge.

So, I hope this article alleviated your concerns about rewriting your C++ codebase. It can absolutely be done, just pick the right programming language for you and your context, do it incrementally, don't overpromise, establish a rough roadmap with milestones, regularly show progress to stakeholders (even if it's just you, it helps staying motivated!), and make sure the team is on-board and enjoying the process.

You know, like any other software project, really!

⏴ Back to all articles
