Interviewer
What have been your motivations for learning Rust and choosing to use it?

Participant
I got a job at a place that I used it.

Interviewer
Gotcha. Do you have any particular area of expertise in other languages before that job?

Participant
Yeah. I'd done a lot of C++ before.

Interviewer
Gotcha.

Participant
And C. Yeah.

Interviewer
Gotcha. More C++ than C?

Participant
Yeah.

Interviewer
Gotcha. Is there any particular details about the types of C and C++ applications you're working on that you can share?

Participant
I was doing game dev and then I worked on a web browser.

Game Development
Web Browser Develop

Interviewer
Oh, okay. Gotcha. So would you say those are fairly different projects architecturally in terms of where you're engaging with the system?

Participant
Not really. I think a web browser is kind of like a weird game engine.

Interviewer
Oh, okay. How so?

Participant
I mean, both have to worry about a lot of graphics input. Their big user

interface applications, they have to meet 60 frames per second. They have a lot of, I don't know, a lot of the same concerns.

Interviewer
Gotcha. So was performance mainly a concern then in your development work in both projects?

Participant
Yeah. Yeah, for sure.

Interviewer
Gotcha. Were there any other primary concerns? Like, did you have any large-scale memory safety issues?

Participant
Definitely in a browser, for sure, in a game engine, a lot less so.

Interviewer
I guess in the browser, were there any particular patterns that you noticed in your work? In terms of memory safety or just for Rust or for other stuff?

Participant
Right now, just for the C and C++. Yeah. I mean, there's plenty of patterns. That's such a vague question that it's hard to answer.

Interviewer
Gotcha. So then I guess for the Rust side of things, when you joined that company where you first started using Rust, what types of projects was that used for?

Participant
I mean, initially, services stuff for that web browser and libraries used by that and other mobile applications that needed to get similar sync data, sync data and accounts and stuff.

Interviewer
Gotcha. Okay. So has that mostly been, like, has that been the space where you've done Rust development Since?

Participant
Not really. I also worked on a end-to-end messaging library for a place and I've done a lot of database stuff since then.

⋮ Networking &...buted S

Interviewer
27 Okay. Sure. Yeah. So then another broad question, what do you use unsafe Rust for?

50

51 **Participant**
Performance, FFI, implementing abstractions that you couldn't do in Safe Rust, like extending the language in ways that wouldn't quite work in Safe Rust. A lot of things kind of boiled down to different kinds of FFI, for example, different kinds of syscalls you might talk to the OS with, a little bit of control over data layout and accessing one type as another type. That kind of stuff is also common for using Rust or using Unsafe for other. Yeah, I guess those are probably the big things and there's, like, a long tail of, like, one-off cases where, I don't know, accessing global variables. I mean, interfacing with weird, well, like, okay, C API counts as FFI, so maybe not that one.

14:2 P... Increase Performance
No Other Choice
14:... Transmute
... Static Variables

52

53 **Interviewer**
Gotcha. Yeah. So I guess let's start with the first category that you mentioned with performance. Is that mostly looking at places where there are bound checks and using Unsafe versions of those operations that skip the checks, or are there other particular areas?

54

55 **Participant**
Usually you can get away with, you can avoid the, you don't need to use Unsafe to get away from bound checks because you can use iterators or you can find a way to, like, prove that the bound check is not needed in a way that's, like, safe and exposed by the library. But sometimes, sometimes, and I mean, bound checks are usually pretty cheap because they're correctly branch-predicted almost always. It's more about, like, I guess the most recent time I did some Unsafe would be doing SIMD intrinsics, like, that was earlier today, I did some of that, so, yeah, and that's definitely a performance thing and also, yeah, Unsafe because of target, just me. Not necessarily matching up.

14:5 Usually you can get aw... Increase Performance
14:6 I gu... Intrinsics

56

57 **Interviewer**
Gotcha.

58

59 **Participant**
I'm also just implementing data structures or algorithms that require a level of control over how things are stored.

60

61 **Interviewer**
Gotcha. Tell me a bit more about those.

62

63 **Participant**
In particular, there's a lot of, like, level of atomic algorithms that you might want to do and having precise control over the, like, data layout allows you to be able to store something in a way that can be accessed in a lock-free manner without causing issues. That's a common one and the compiler can't necessarily prove that. I mean, some cases you can be fine with without Unsafe, but, like, for example, if you put something in an

14:7 In parti... No Other Choice
14:... Box<T>

atomic pointer and put a Box in an atomic pointer, you have to put a raw pointer and then when you get that out to use it, you need to either, like, turn that pointer into either a reference or a Box or something else and that is Unsafe. Yeah.

14:8 I me... Box<T>

Interviewer
Gotcha. So, you're accessing these, I guess, could you describe one of those scenarios where you are?

Participant
Yeah, I was like, Queue node, nodes in a lock-free queue, it's very simple.

14:... Unsafe Data Structure

Interviewer
Gotcha. And then, related, you also mentioned that one of the key uses you have is implementing data structures and primitives that go beyond what Rust guarantees and library provides Typically. Is that related to this use case or are there other examples that you can think of where that happens?

Participant
A lot of the time when I'm doing that, it is for work on the standard library, which is one of the regular contributors to that. But, because usually, personally, I kind of just, I don't know, I don't, when I publish libraries, it's not usually like a new fundamental unsafe assumption. That said, I can think of a couple examples that I do use. One of them would be the yoke-able crate that allows self-restructural data. One of them might be, there's a lot of failed versions of that, but ByteMuck allows safely transmuting different data, viewing one type as another type, in some cases, in a lot of cases, you can't really use ByteMuck, and so you're still off to using unsafe. But, yeah, yeah, that kind of stuff, like, a new fundamental operation that, you know, the language doesn't necessarily provide.

14:... Rust Project Contribut

14:53 One of them... Data Processi...& Seria
No Other Choice

Interviewer
Gotcha. Could you give an example of one of those?

Participant
Like, having access to self-referential data is a big one. Yeah. Or there's a lot of more operation-shaped things that you can implement a lot of stuff in inline assembly.

14:54... Inline Assembly

63 **Interviewer**
Gotcha. Gotcha. And when you're doing this, you mentioned that, in particular, the self-referential structs, that there have been a lot of attempts that have gotten that wrong.

78

79 **Participant**

Yeah.

Interviewer
Were there any particular reasons why, or?

Participant
Really hard. I mean, Rust really assumes that you don't have that. And if you're until relatively recently in the scheme of things, there wasn't PIN, and now PIN exists, but it's very hard to use correctly. And especially if you take a reference out of PIN, it no longer is, I mean, it's still PIN'd, but it no longer necessarily could be, you can, either it's immutable or you now use it in a way that violates the requirements of PIN.

14:10 And if you...

Pin<T>

Interviewer
Gotcha.

Participant
It's kind of a bit of a mess. Or it's an un-PIN type, in which case it doesn't have self-referential data.

Interviewer
Gotcha. Okay. So was that the key problem, people using PIN incorrectly?

Participant
Or PIN not existing, I think, because I'm thinking of like the rental crate and, God, I don't remember the one that was before rental, but there was another one. And those both are pretty broken, known to be pretty broken. Not all uses of them are broken, but definitely, I don't know, it's not something I would recommend anybody use now.

Interviewer
Gotcha. Gotcha. So then the FFI usage that you mentioned with Unsafe, what types of scenarios does that come up? You mentioned in particular like system calls, but are there cases?

Participant
Oh, yeah. Very common one is like calling into C libraries. And then the other way is exposing Rust code to C or other languages. I did quite a bit of that in the past.

14:12...

Directionality

Interviewer
Gotcha.

Participant
79 And I still maintain a bunch of C libraries. Bindings for C libraries.

Interviewer
Gotcha. Okay. So let's start with the sort of, like are there any cases where the FFI is bidirectional, or is it mostly just been unidirectional between Rust and another language or vice-versa?

Participant
I mean, it can be bidirectional in the sense that I give C a callback and it calls it later. Or it could be, I mean, kind of the fact that it's tricky to do mutually recursive dependencies and have it link all correctly, that kind of stops that. Also, I don't know, usually when I write a program, it's going to be one way or the other.

14:13 I mean...

Bindings are a Fiddly M

Directionality

Interviewer
Gotcha.

Participant
And if I'm writing a library, it's probably going to be a Rust library rather than something else.

Interviewer
Gotcha. Okay. So those Rust libraries that are binding C, or just any particular foreign language, are there common issues or patterns that you found in the differences between the memory models of C and Rust that?

Participant
Yeah, there's a few. One is particularly around thread safety. I've found that while Rust enforces thread safety based on, like, types and, like, this type is send or sink and that means all of the operations on it can behave a certain way. C tends to do it based on functions. So you'll have this function is safe to call on multiple threads and this other function, even if it's within the same type, even if it's like a method or conceptually a method because it doesn't, yeah. And that is a pretty annoying thing to, like, work around in terms of binding because you kind of have to, like, just be like, well, shit, this is not quite thread safe, even though, like, 90% of the functionality is or you just don't expose that function or something because you probably don't want to take a lock even in RwLock. But yeah, that's one case. Another case is C will often have kind of a bit of a rat's nest of pointers in terms of, like, an object graph. And that doesn't quite, I mean, if you can have, like, one owner of all of the objects, that works fine, but it is kind of like a mutually owned situation that sucks. Or even if you do have one owner of the object, you often would want that to be stored on, like, the same abstract, like, stored in a struct with the object graph and have that be, like, a common abstraction. But then you have a case where, like, that's effectively self-referential because you have the object and then you have on the same type that has, like, data owned by that object.

14:15 I've found that while Rust...

Different FFI Memory I

RwLock<T>

14:14 Another case is C will...

Different FFI Memory I

113

114 **Interviewer**
Yeah, sure.

115

116 **Participant**
So it's referencing the other field. That while it's not, like, concretely self-referential in terms of this is now, like, an object where if you moved it, you'd get dangling pointers, it still doesn't get expressed well in rust. And so you can see stuff like the get to crate works around this by having just a kind of a shitload of lifetimes on everything. And, like, that's a great crate. And it's a really good approach. But it also is very hard to, like, put those objects, like, use those objects in a way that's not, like, okay, I just have one function that does all the get to stuff here. And then, like, I throw away all the objects at the end. And then it's hard to keep those objects around for very long unless you want all of your structs to end up with a bunch of lifetimes, which is okay, but is painful for a, you don't want that in a library and a lot of, it'll scare away new programmers for sure.

14:16 But it also is...

Difficult FFI

117

118 **Interviewer**
Gotcha. So one aspect of this that I'm interested in is in my limited exposure to some rust and C binding crates.

119

120 **Participant**
Mm hmm.

121

122 **Interviewer**
Like one example I'm thinking of is [library].

123

124 **Participant**
Yep. Yeah.

125

126 **Interviewer**
So with that, you have this single global state object that you have a pointer to and then that's access to the FFI, but you and you have some self referential patterns going On. But that's mostly constrained to the C-side in a way that if you're just engaging with the rust side or even like the just the internals of the rust bindings, you never really have to worry about that. So the way you're describing this with the object graph becoming a problem in terms of rust memory model, like, is

127

128 **Participant**
“there just more of a problem if you, if you were handing out stuff to the object, for example, if you have that, like, ob, for example, the [name] create it has a font context and it provides a font that references it. And now internally, it does actually have like a database of like the fonts. And so that's why the font references it. And that's kind of awkward as a result.

129

130 **Interviewer**
Gotcha.

131

132 **Participant**
Because you can't then put like, be like, Oh, this is the thing that holds all the font stuff and it has the font context and then the font, which you probably want to do in a lot of cases. Because I don't know, you probably don't want your whole program to now have a lifetime for font stuff. Yeah.

14:55 Be...
Difficult FFI

133

134 **Interviewer**
So it's this, is that generally the common case where you have this sort of central store?

135

136 **Participant**
It's tough because for that, I would say that case should probably use like some sort of weak reference on the font object, maybe because I think, but there's other cases where that's not appropriate where you actually do want the lifetime because it's a short lived object and having it be a reference counted thing would be much more trouble than it's worth, especially if the parent object is send, which is note the case for [library] stuff. As soon as you have like a reference on it, you now need to track, as soon as you're doing reference counting where you have like the child object, not containing a lifetime to the parent. And you have to like make sure that one is not used on a different thread than the other, just a huge pain. Yeah.

14:17 It's tough be...
Send
Weak<T>

14:56 As so...
Difficult FFI

137

138 **Interviewer**
Gotcha. Gotcha. So is this, you're mentioning two patterns that seem to have different use cases where in some cases it's appropriate to use a lifetime. In some cases you would want reference counting. I guess, is there an opinion that you have on the design of some of these crates and how they're handling the object graph from C where you feel like it just needs to be better? Or that there's like a...

139

140 **Participant**
Also my own crates. I mean, there's a lot of code that needs improvement and I don't know, you just kind of deal with It. Like, my code's not perfect either, but like that's one that I definitely feel, I mean, they might have changed it since I last used it. So maybe that got better. That's one that I definitely was like, ah man, this is way more effort than I want to put into the fonts. Yeah. So I just made that one static. Transmuted that one to static, which is another use case for unsafe is working around a bad API when you know that you can use it correctly.

14:18 That's...
Easier or More Ergono
Transmute

141

142 **Interviewer**
Yeah, sure. Yeah. I think I've definitely encountered that case of having like your single global context object and then individual pieces of the

graph you're handling in Inkwell, the LLVM language defining crate where you have this single context object that everything has to reference and it ends up being this nest of references.

Participant
That's closer to how rustc is. I think in that case, it's probably fine. Like in rustc, it is like everything references the type context because it has a bunch of arenas.

Interviewer
Gotcha.

Participant
And that once you get used to it is pretty good. But yeah, I mean, it definitely having those lifetimes is awkward at first. For that case, you're never going to really abstract away from, you don't necessarily want to like have this single unit of like conceptually related functionality where it's like the font context and the font object. That doesn't make sense to abstract away for something like LLVM or rustc in terms of the module object or LLVM or the like type context in rustc. Maybe it wouldn't be the module object. It's been a while since I used LLVM.

Interviewer
Gotcha. So I guess I'm just trying to get the difference. I guess with that structure, it seems like it's almost more of a tree in this case and that I have the global context, which is a set of modules and then those modules have sub references. So I guess there isn't necessarily like a lot of...

Participant
I would imagine it's a graph. I mean LLVM values are graphs like internally and because they all reference what they're used by. And certainly in rustc, it just has a bunch of arenas so that it can easily do graph stuff, which is why everything has the lifetimes.

Interviewer
I guess I'm thinking more in terms of the way that I'm engaging with the API versus something that is very explicitly cyclical or explicitly self-referential. Because at least in terms of the way that I'm engaging with it, I never have to touch that. But in some of the patterns you described, that seems like it would be more common.

Participant
Yeah. I mean, definitely I would say that probably it's pretty hard for somebody to implement inkwell under the hood. It sounds tricky. I
142 definitely wouldn't want to be implementing the bindings. It sounds very involved. So yeah. And I imagine that probably part of why it's involved is that you have to figure out all the lifetimes for everything. Definitely when

I've used LLVM, I had no idea when stuff was freed. I feel like they don't document it.

Interviewer
Yeah. I know that. I have yet to see documentation on that in the parts that I've looked at. But then again, I'm also not like... I have engaged with the documentation is enough for me to comment on that. But I guess my question is, are there other patterns that you've seen you mentioned like that you feel are problematic in how references are exposed to...

Participant
Yeah. Another one that I actually don't have a good solution for is C APIs where they either take a source in a destination and they do allow the destination and the source to be the same light pointer, basically, or like a slice. They do a transformation on something and then they allow it to be in place. That is really hard to model well in Rust, unless you... Especially if you need a bunch of setup to have... And you don't want it to just be two separate APIs where you boil down to the same underlying pointer operations. It is especially bad to try to re-implement part of that in Rust. I don't know exactly how to... I have an idea of how I would do it, but it sucks because you'd have to use Cells a bunch and would not feel idiomatic.

14:19 Yeah...
Different FFI Memory I
Difficult FFI
Return Pointers

14:20 Especially if...
Cell<T>
Different FFI Memo
Easier or More Ergo

Interviewer
Gotcha. Okay. Is that more of the challenge?

Participant
That is a challenge. For sure, managing alias saying and making sure that you never have a mutable reference in that could overlap any part of a shared reference. That's why Rust is hard to write on safeguard for almost. Yeah. Because it actually isn't just references. If you take a reference and you get a pointer and you turn it into a pointer, then that pointer still has to follow the rules of the reference, even though it's not checked by the compiler anymore.

Interviewer
Yeah. Yeah. So is Miri a common development tool then that you've been using in these situations?

Participant
Miri is viable if you're not doing FFI. If you're doing FFI, you just have to hope.

14:...
Miri doesn't s...ort this
Wants Solution

156

170 **Interviewer**
I guess, are there any patterns to the FFI usage that you look for and investigate because they are potentially barring violations?

171

172 **Participant**

But I guess in my thought, the way I'm thinking about it. Yeah. So it's a somewhat common pattern for people to... I don't know that this is actually something that will bite anybody. It's somewhat common for people to... Rather than using a raw pointer to wrap some FFI type, they'll try to make it a reference and just pass that around. The reference is zero size and it's a way of emulating an extern type, which isn't a stable feature yet. And that's actually unsound under our current models. So that bothers me when I see it. There's definitely a little bit of a red flag that the person is getting to. I don't know. They're trying to be too clever with it and I'm a little bit skeptical of the rest of the stuff. Another one is any code that does callbacks into cases where you give a Rust callback to C. It's really tricky to get... I think we've gotten better in the past six months or so, but it's generally really tricky to avoid issues around panicking. And by gotten better, I mean, if it goes wrong, now you abort rather than it just being undefined behavior. I was curious. It's very hard to catch a panic in a way that can't panic itself. And even if you catch it, then what do you do with the panic code? Are you going to store it in a global variable? I mean, that's viable, but it's not great. You could store it in a thread local, but accessing thread locals can panic. Yeah, it's a huge pain in the butt. Another place that I look out for is any code dealing with manual allocation. That's super hairy to get right in Rust for whatever reason. I feel like our layout struct, it's neat conceptually, but it's very easy to misuse and very hard to use correctly. I don't feel like it was the right abstraction, but it's there and we can't get rid of it now.

14:22 Rather than... — Code Smells; Shifting Ground

14:23 An... — Difficult FFI

14:58 Another... — Code Smells; Layout

173

174 **Interviewer**

What would a better abstraction be to replace layout?

175

176 **Participant**

I feel like actually just size and align separately because now that you have layout, it tries to encapsulate all of the safety requirements for passing something to allocation fun for perhaps. And it tries to be somewhat self-consistent in terms of like, well, the size needs to be able to be rounded up by the alignment. Otherwise, it's undefined behavior, but that doesn't quite... It doesn't handle all of those cases and we've had to adjust it a couple of times to tweak the safety requirements, which feels really bad because that makes some code that was written correct when it was written into code that had undefined behavior and I don't know. It's just very easy to accidentally cause integer overflow when trying to compute the size and the alignment or very hard to avoid it in a way that is provable. Also, it doesn't handle some cases of, for example, in Windows allocation APIs, there's the ability to allocate within the C runtime rather than the kernel DLL, but the ability to allocate a three-parameter allocation function, which takes an offset, an alignment in the size, and the offset is where you can have a header where after that many bytes, it is aligned to the thing. And that's a really nice function that you cannot quite express with layout without having a bunch of extra overhead. But I mean, that's not that big, but I don't know. Yeah.

14:24 I feel like actually j... — "Safe" API; Layout; Wants Solution

14:25 but that doesn't quite... It doesn't han... — Difficult to Encapsu; Layout

177

Interviewer
Gotcha. And I guess in line with that, you mentioned using every single one of the memory container types we had in the survey and on safe contexts, so I'm just going to ask the question.

Participant
Yeah, I've been doing Rust for a long time.

Interviewer
Yeah, from the years, I'm like, yeah, that checks out to me.

Participant
Yeah.

Interviewer
But I guess let me sort of explain what I'm trying to get at with this question. Maybe that'll help. So the problems that I'm most familiar with as pitfalls are with Box and Rc where if you unwrap them, you need to ensure that you're still eventually deallocating that memory somewhere and that you need to make sure that if it's allocated in Rust, it's deallocated in Rust. Are there other pitfalls with memory container types similar to those or related to those?

Participant
I mean, definitely deallocating with the... If you allocate memory with one layout, it has to be deallocated with the exact same layout. That's a big one that is a real pain in the butt and it's very easy to get wrong and coming from C, which doesn't give a shit at all in terms of... I'm sorry.

14:26 I m...
Different FFI Memory I
Layout

Interviewer
No. Be as raw as you want.

Participant
Yeah, okay. It doesn't hear at all in terms of what you give to free. Yeah. Even posix memalign or Aligned Allocation Functions. Yeah, just throw it into free, which is very different. That's a big case. Definitely making sure that memory that you allocate, perhaps not with Box, but in general, if you're manually allocating stuff and then giving it into Box, making sure it's actually initialized and in general uninitialized memory is something I failed to mention in terms of a case where you use unsafe code. But it is one and it's one that's very easy to get wrong.

14:27 Definitely...
Box<T>
Box<T> Bug
Using Uninitialized Me

Interviewer
I guess have you found that MaybeUninit has been helpful there or are there problems related?

Participant

Yeah. It's tricky to use right, though, because you often want to be able to have a function that takes it MaybeUninit, like a mut MaybeUninit, and then initializes it. And you want to be able to like, okay, I actually have an initialized slice and I want to pass that into that function or something. And you can't do that because you can't really trust the function you're passing it into, unless you know the source of it. But you can't trust it not to just write maybe uninit into your previously initialized bytes, which now would be unsafe. That's more of a, that's like why we couldn't replace like the read APIs with a maybe like mute maybe on the slices. That's why there's like this, I don't know, there's a cursor type for uninitialized, a partially initialized slice is very complicated. But yeah. I don't know, like initialization is a big one. Reading in the right context is pretty important, although in practice it, I mean, it's a little bit of a theoretical issue because nobody bothers setting a custom allocator. Although it definitely is something that people should get right. There's allocator in the C allocator or something, I don't know. Yeah. I think that's about it. Or, oh, no, for Boxes, people assuming Boxes just a memory like a heap allocated pointer that's automatically freed rather than something that must be unique. Although that might change soon, but yeah.

… MaybeUninit<T>

14:30… Shifting Ground

Interviewer
Gotcha.

Participant
I mean, it's not clear if it will or not, but it seems like the winds are pointing that way.

Interviewer
Okay. I guess one thing as well here with Box that's been mentioned in prior interviews is that there are cases where you have pointers into a Box and then the Box is moved and it validates those pointers.

Participant
Yeah, that's kind of what I mean in terms of like it must own the value. I think that's terrible.

Interviewer
Okay.

Participant
Not a fan of that at all. I think that in general, a lot of the things the compiler is allowed to assume about Box are not that useful because we already get to assume those things about all pointers that get written to or read from because we get no alias as soon as it turns into a mut or shared reference and I don't know. That's almost everything has that happened all the time. So putting it on Box doesn't really help very much and definitely causes a lot of foot guns.

14:31 a lot of the t… Wants Solution

196

Interviewer
Gotcha. Okay. Gotcha.

Participant
Like that is definitely a problem.

Interviewer
Gotcha. Okay. Are there any foot guns you can think of related to the cell types?

Participant
I mean RefCell is pretty panic prone in practice. But that's not that big of a foot gun. I don't know. I definitely prefer people use the cell types and use like global mutables and stuff like that. Yeah. I think the cell like sell itself is probably a little bit underused because you often want like just a counter or something. Or the atomic types. If you need to be. Yeah.

14:65 I mea...
Cell<T>
RefCell<T>

Interviewer
Gotcha. Gotcha. Okay. So I want to kind of step back to your work as a library developer. So there are sort of two aspects of Rust libraries that I'm particularly interested in the first I think is in documentation. So in the API is that you're exposing to users. I know that with unsafe. So there are sort of two usages where in one case it's like you need the unsafe operation. And then in another case, maybe what you're doing is not necessarily heavily unsafe, but there's some sort of burden on the person who's calling that API to have some sort of precondition for it to be safe. So you're talking more about those situations in your code.

Participant
Yeah, definitely. That's definitely basically how I think about it. You either have like you need the unsafe documentation on the call to prove that you are upholding the the requirements that are documented down the like the function itself. That said it's kind of tedious. It's great for the Rust library, but definitely for my own stuff, I find often will be like, well, this is an internal function. It's like just gets a line or something. I see. But yeah, I mean, it definitely helps because it makes it a lot easier to audit, but it also can be very hard to especially for FFI to actually pin down the safety requirements. Like the [library], for example, there's this VFS extraction, which is literally the whole interface to all memory allocation and the OS and everything. And like, I don't know, I was trying to come up with a way to describe explain the safety requirement and it's really just don't fuck it up if you try to write one of these. Like you just like read, read a bunch of the examples and like the ones that are enabled by default, like, please, because it's very hard to do. And that's that feels bad to like have no concrete safety requirement beyond like, don't fuck it up. Although it does kind of like explain the truth of, I don't know, it's at least not bullshitting somebody and saying that. Oh yeah, it's only these like four things, which is way it's sometimes you use a library and you're like that, especially the FFI one and you don't, I don't know. It'll be like, oh yeah, this creates a

14:33...
Documented...ract or I

14:34 Like t...
Difficult to Encapsula
Documented...ract or I
Exposing an U...e API t

like a graphics context or like a GPU thing and like has like a one line safety requirement of like you need to. I don't know, give the data can't have any NANDs and it or something. It's like, that's definitely not all that's there.

Interviewer
I see.

Participant
Yeah. Or like, [library], I think doesn't really explain fully. Because that's unsafe because it can run global constructors from C automatically when you load a dynamic library. And also it's unsafe because it will very like, when you close that library, which it will do automatically for you when you drop the the object. That library may have registered like callbacks to happen on exit. In fact, it probably did, especially if it's a Rust library because it would have had like thread local stuff that the standard library does. And so like that'll probably crash or something like that kind of stuff. It's very easy to gloss over that because it doesn't feel like it's your bug. It feels like it's somebody else's bug.

Interviewer
Gotcha. Okay. So it's this sort of issue where in some cases, you like, I guess that the worst case I can think of is you have an unsafe API that's exposed and it's one that a user of the library would need and there's no documentation.

Participant
But then there's also that can be the worst case scenario. It can also actually the worst case scenario is if it's the document is there and not isn't complete and gives you like kind of or is straight up wrong, I guess. But usually it's, it's more often not complete rather than wrong.

14:35 But...

No Docs > Incorrect D

Interviewer
And then I guess in the second area I've been thinking with libraries is in how you're managing unsafe, both in terms of like encapsulation as well as just how you think about like ratio of unsafe to safe code in your in libraries is

Participant
In the API or the implementation.

Interviewer
I guess, let's start with the API, but then I also the implementation.

Participant
I definitely will try to allow for everything that somebody will want to do to be doable with safe as much as possible there sometimes they'll be like

one or two things like yeah this is like something that starting up the library that's often just like yeah stuff is set up right in some ways like maybe this is going to map a file or something like yeah like make sure nobody's going to fuck with that file or something like that. Yeah, or you're only doing this once or something although that's something you could enforce but Yeah, so that's, but then it might not be the most efficient way to do it and then I'll do give access to either like low level details or a more performant option with unsafe. And like, in particular, handing out the C objects that I won't give that as an unsafe function but I will give that give them the raw pointer to the C object and like, they're going to have to use unsafe to use that. Or perhaps the other way where I allow you to create a, a objective in the binding library from the raw pointer that it that has to be unsafe. And also probably that's like a bit of a headache because you maybe don't want them to. Maybe you don't want to drop that when you're, or free that when your object goes out of scope or something that's. There's a few ways I have ideas for doing that but it doesn't come up enough for me to really worry about it much. Yeah, so definitely performance is like the number one reason for having like an unsafe API like the best reason to have like, this is the same function but without checks. But yeah, the other one is probably like, I don't know, just just accessing functionality that you can't really find a way to expose safely or you just don't have time to expose and it's just like hey, yeah, use the C API.

14:36 . Yeah...
Exposing an U...e API t
Increase Performance
No Other Choice

14:37 An...
Exposing an U...e A

14:38 There'...
Exposing an U...e API t
Increase Performance

14:39 But...
Easier or More Erg

Interviewer
Okay. So, in this case, it's, it's more you have the option of using the C API directly or maybe a version of your Rust bindings without the checks but you have that as an option to the user for performance. And then you have a safe encapsulated version for if you're really just concerned about about.

Participant
236 Yeah, and I mean, ideally the safe version has very little overhead. It's pretty rare for there to be none. I mean, somewhere you're going to access an array and it's going to have a bounce check and that's just how it is. Or a slice but yeah. But that's very cheap and not worrying about that took me a long time. Yeah. And then in terms of the implementation, if, yeah, I don't worry about the amount of unsafe and the implementation particularly and I do worry about the way I think about it is I want each like thing I'm doing the unsafe to be easy to verify on its own. The worst case for this is something like a garbage collector where the implementation is correctness depends on the algorithms correctness itself. Like that kind of thing seems like a nightmare to audit because you have this huge thing and you need to implement like make sure the whole algorithm is implemented correctly. In order for any of the memory safety of the system to be verified, whereas a lot of unsafe code is pretty like, okay, this is like doing these three checks. That's exactly the thing that it needs to check or like this is turning a slice of the U32s into a slice of U28. It's like, yep, go for it. Yeah. I don't know that kind of stuff. I don't think I don't worry about it all anymore. I'm going to be nice if like it were built into the standard library and I'm doing it a lot. I'll reach for something like

14:64 Yeah,...
Preference for Safety

14:40 The wors...
Unsafe is Diffi...to Und

14:41 Tha...
Transmute

ByteMock, but yeah.

Interviewer
Okay. Okay. So I guess moving towards foreign bindings in a bit more, you've used different variants of bindgen but then also CXX, autoCXX and UniFFI. Could you describe the usability of these tools and the extent to which you have not done a ton with autoCXX.

Participant
That's something I've been using like playing around with recently. CXX kind of is a pain in the ass to use. In general, I kind of think that's an inherent problem with talking to C++ and I don't know. I almost am on the side of just like writing a small little C library just early like a C header and like C++ implementation file to like just call into stuff. But yeah, bind gen I think is a little bit it for a certain type of it library. It's really the only option, but I also don't think it's that good. Like it gets a lot of stuff. It's very easy to to misuse. It's very easy to think that you're generating bindings for that you can like check into your repository and reuse. And it's so easy to get that wrong if you don't know C like for a long time it actually would generate size T that depended on your platform rather than just using U size. And so a lot of code had checked in bindings that were just wrong on Windows 32 or Windows 64 where long is not the size of a pointer 32 bits. And so like you would have size T equals C long or something. It's like, well, that's wrong. Now I complained about that. Now they default to using U size. Although technically that's not guaranteed by Rust. So my break time on some systems that probably that's the code doesn't work on. Yeah, other than that, Uniffi was cool…But it, I mean, you got to buy into like writing a separate IDL for it. And so like it's a very limited use case. But it's cool. I can generate like a bunch of different bind it like bindings for a bunch of different languages. If you had, if I had a library, I don't know if Uniffi still maintains. So assuming even if its still maintained, which probably is not true. And I had a library that I wanted to expose to a lot of different languages I would consider using it. And it didn't have async because I don't think they ever figured out how to do async with that. I guess is another part that like makes FFI hard. Other than that, let's see C bind gen. I don't know. Do people still use it?

14:42 CXX ki…
CXX
14:43 But yeah, bind gen I think…
Limitation of Bindir
14:62 Ye…
Idiomatic FFI Encapsul
Uniffi

Interviewer
At least I've from the people I've talked to it seems like it's still being used.

Participant
I have not heard of people using it in a while. I just read headers if I need to do that. For like weird languages for like, I would definitely use it for PyO3 over handwriting Python bindings. I definitely use, I don't remember what the node one is, but like there is a node one. I work on one for Postgres for exposing like Rust functions as SQL. Like there's a lot of that kind of stuff where it can really make it a lot easier. C sharp is not so bad as long as you are C sharp and it's not so bad as long as you're like exposing it as a native extern. Java can be okay if you use JNA, not JNI. I would definitely not use JNI for it because it's a nightmare now. Although I

14:…
Generation VS Validati
14:…
JNA

think they've, there's some new Java thing and latest Java that's supposed to make FFI better. I haven't checked. I've not worried about that in a while. I don't know. Usually there's a way to make the two languages talk by going through C at worst.

Interviewer
Gotcha, gotcha. And then I guess in terms of other development tools but more generally for bug finding you mentioned you've used, you've been using the sanitizers. Clippy, Miri, could you describe how those have been helpful for you in development?

Participant
Miri is very cool for the set of things that can run Miri and don't take too long to run Miri. And the sanitizers are extremely good. At least thread sanitizer is absolute godsend. The biggest issue with it is it doesn't support fences. And so you have to like rewrite your atomics in terms of like loads on the atomic object rather than an offense. That's kind of a good idea anyway for performance on a lot of chips. Like on AR64 these days, it's faster to do an acquire load on an atomic than it is to do an acquire fence. But they have slightly different semantics so you do have to watch out for that. The fence acquires everything that was previously loaded in that thread whereas the load only acquires that one value that you loaded. Thread sanitizer is the big one. AddressSanitizer, I've never actually really had it catch anything that was a big bug, but I don't know. It's nice to like run it on CI and be like, yeah, great. I definitely have that on several of my libraries. I've also used Loom for like currency stuff. There's a model checker, like a port of a CDS checker to Rust, although it's like 80% of a port and it changes the architecture. So fixing the last 20%, which you desperately want to do after using it for a while, it's intractable. I wish it supported CXT correctly, but it does not. That's one of the ones people use all the time and it's the hard one. But yeah, another... It's a little bit of a different use case. I had a decent amount to debug memory usage issues and stuff like that. Yeah. I don't use Clippy very much, except unless I'm half due for a library. I kind of just stopped using it one day and been way happier about just this really annoying. A lot of those lints don't help. Yeah. I like the library in the code and I think the fact that Clippy made the Rusty linting API a lot better. The results are always very helpful. It expects you to allow a lot of lints, which sometimes works and I don't know. It's just often very annoying. I think I'd use it if I expected to work with a lot of people who are very new at Rust though.

14:46...
Wants Solution

14:60 Ad...
ASAN

Interviewer
Gotcha. So then are there any particular problems that you face in your use cases for unsafe where development tools haven't been able to solve them, but you feel like there should be something to help you?

Participant
I mean, it'd be really cool if somehow Miri could also have the sanitizers back end for having every syscall implemented. Not every syscall, but having a large amount of libc implemented in terms of you could just call

14:47...
Miri doesn't s...ort this
Wants Solution

in to see libraries because it does interposing and stuff like that. There's some plan for using libffi to integrate with Miri, but it never was really clear how that was actually going to be able to detect any... Most of the issues you're worried about are cases where C does something that you are... Which C writes to a reference you have in Rust or something or C uses memory after it's freed and Miri probably can't even catch using the memory after it's freed, which is the most basic of things that you want. But yeah, a lot of stuff is being worked on. I think a case for more debugging tools around global allocators, it's very hard to use global allocators correctly. I feel like everybody messes it up constantly and doesn't notice, but I don't know. I was going to work on that at one point and then you get busy.

⋮ Miri doesn't s...ort this
Wants Solution

Interviewer
Gotcha.

Participant
You can just write a global allocator that just checks a lot of stuff, but yeah, it's a hassle.

Interviewer
Okay, interesting. And then I guess for my final question is a pretty broad one too. Choose sort of your favorite most tricky bug that involved unsafe Rust and describe how it manifested and what was your solution for fixing it.

Participant
That's tough.

Interviewer
Or just any bug if you don't have a favorite.

Participant
A lot of the really tricky ones end up being like, ah, it was the linker this
time and it's not really unsafe. It's like unsafe in a sense and it causes a
crash, but it's like, I don't know. Yeah, it doesn't have a satisfying solution
other than I fixed the linker flags or like I made it so that I don't need to, I
don't know. I reconstructed the build system, so it didn't need to do that
stuff. I guess. I mean, definitely one of them has been... I mean, a lot of
these are fixed now, but like the handling panics, it took so long for me to
figure out how to handle like wrap a function and catch a panic and then
not have a case where that function could panic with a payload that itself
when it was dropped panics again. Which is a really ridiculous case to
worry about because who is trying to like really hard to hone their own
program, but... Yeah, I'm not sure I have a satisfying answer for this. Too
256 many small bugs and not enough, I don't know. Or a lot of them are just
like, oh, there's a really hard threading issue where I just got the ordering
wrong or like didn't realize that as soon as like there's a reader still
around holding onto this like no object when I freed it because... Yeah,

14:59 I mean, a lot... Panic Safety

14:48 Or... Concurrency Bug
Data Races

the kind of stuff is very common but like isn't... The solution can be very hard, it can be like read your whole algorithm or it can be just like do some fencing or something. Okay, that's it. 268