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

Participant
Oh, that's a good question. So primarily, I was the C++ developer for a long time. Well, I am a C++ developer. And so typically, I used to really enjoy doing C++ in my free time, because I worked as a web dev. So I mostly did things like Node and Ruby, and so on and stuff like that. And so I finally got some C++ jobs. And I don't really like using my work languages for my home projects. So I was like, all right, you know, I hear Rust has co-routines. I don't know anything like IOuring or something like that. Oh, because I was like a networking dev. I did a lot of like Azio and C++. So I was like, all right, let's give this a shot. I'm going to learn Rust co-routines and IOuring in Rust, you know, just for the roles. And so that was what really drew me to it. My first project was re-implementing the standard library's vector, but as a single pointer. So that was my foray into the language.

Interviewer
That's a pretty big starting point there. That's something on like my to-do list to look at. So you mentioned you were a C++ developer for a while. Did you have any particular challenges accommodating to Rust's memory model versus how you reasoned about memory in C++?

Participant
Yeah. So one of the things that's really different and the problem is there's so one of the biggest problems I think with unsafe Rust is there's no actual specification of like an object model. So like right now, if you're writing unsafe Rust and you want to. So one of the big things with C++ is you have the decoupling of object storage from object lifetime. So, you know, C++ goes through a lot of wording, like very careful wording to be like, you know, we have the notion of storage and installation and then there's objects whose lifetime live within that storage. And so that's when you have things like placement new and then, you know, in place destructors. And so with Rust, there really is none of that.

9:1... Shifting Ground
9:49 So one of the... Feature Disparity

Participant
So my first project was implementing vector. And so when looking at vector, one of the first things you're doing, you're doing in place construction essentially. So the problem is there's no documentation for that. So all you really do is you kind of just pull up GitHub, you look at what the stdlib is doing. And you're like, oh, okay, if I just like randomly memcpy a region of storage, it like gives you a valid object, you can start accessing and dereferencing. It's also very hard to know. So like in this little runtime, this IO runtime, I'm writing, it's, it's very hard to know when it's safe to turn the innards of like an UnsafeCell into a mutable reference. Because it's so strict. And the problem is you can't run a tool like Miri. I don't know if you've heard of that. It's like a little MIR interpreter. Yeah, the problem is you can't run that because you're using FFI because if you're

9:50... Shifting Ground
9:2 So lik... UnsafeCell<T>
9:3 A... Miri doesn't s...ort t
Wants Solution

touching real sockets on the system, you know, Miri's not going to work. And so you're like, you have this whole runtime run and you're like, God, I hope I'm not violating their aliasing rules. Yeah. But just because Miri's limited to the MIR, it's like you just can't use the tool at all. Not well, theoretically, if I was motivated enough, I think I could pull out segments of my code base and run Miri through a subset of the unit tests. But as terms of like a comprehensive, see, the problem with that is it leads to like an awkward constraint design, where sometimes you really just want to call the FFI function right as you're forming a mutable reference. I guess maybe it's an architecture problem. But yeah, it's definitely not painless to integrate Miri into a code base using real FFI.

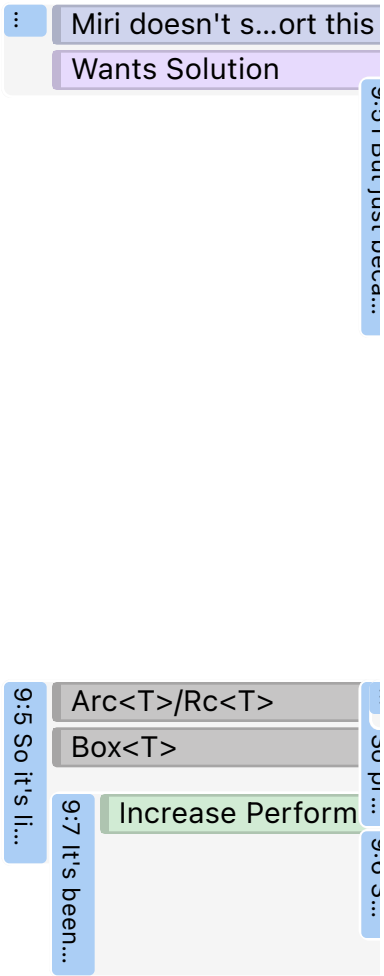
Interviewer
Gotcha. Yeah. So what I guess then moving more into the unsafe rust area, what have been your uses for unsafe?

Participant
So primarily, it's really just been for dealing with the raw system calls. So it's like raw allocation, raw deallocation, which is always nice. Pretty much leaking and unleaking the smart pointers. So like RC and Box, stuff like that. Most of it's primarily motivated by FFI other times. It's been simply for performance reasons. So like I tried doing a little bit of image processing coding in Rust, too. And I had someone help me do some manual SIMD intrinsics in there. So I mean, you definitely need unsafe for that.

Interviewer
Gotcha. So you have the FFI stuff, you have the SIMD intrinsics, any other uses?

Participant
Not that I can think of. Otherwise, I typically try to avoid the micro optimizations. Like someone showed me if you want to do like a super efficient, like, you know, get the active member of an enum kind of thing, you can have your blanket part of the match statement use like unreachable unchecked. And so the compiler will optimize out of all check because it's like under the perjury, like under the penalty of UB, you're basically saying, Hey, this is the active member of the enum. And if we match, and you know, it wasn't valid in UB. So I saw that. I thought that was pretty cool. But typically I try to avoid that kind of stuff. The benefits really don't outweigh the risks. You're like, Oh, I saved like maybe 10 cycles on an enum match. And then like all like, I was a potential for UB. I've actually been saved by that before where I do some bad copy paste. And I normally just have like a blanket panic. So that catches a lot.

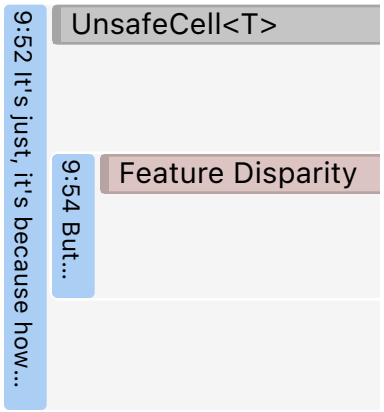
Interviewer
Gotcha. Interesting. So gotcha. So it's more just sort of summarize you mentioned the two use cases being a lot of FFI and somebody intrinsics. And then you're aware of the performance gains that you can do from



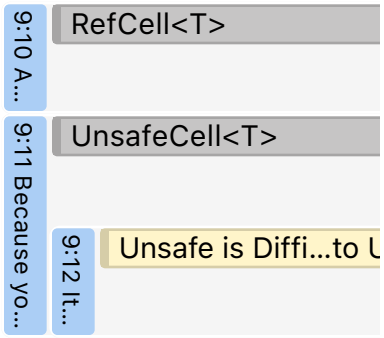
certain tricks like that. But you feel that in general, it's the what you gain from the performance increase is not worth the risk of UB. Gotcha. Okay. So then moving into a bit of some of the types that you use with with unsafe ...

Interviewer
Yeah. Okay, where was I? Oh, yeah. Talking about type usage and unsafe. So there are four types that you mentioned in the screening survey. Let me just pull that up here quick. Okay. So for types you mentioned Box, Arc, UnsafeCell, and MaybeUninit. So let's start with UnsafeCell because I think you mentioned that earlier, there being trickiness with when you can like convert that into a mutable reference. Yeah. Could you describe that a bit like what what are the challenges associated with that and—

Participant
Oh, yeah, sure. It's just, it's because how you want to have to derive raw pointers from the UnsafeCell. And I guess how it was, I think how Rust does interior immutability is really interesting. I know there's like some pithy C plus plus to have who's kind of like, Oh, it's the same thing as mutable data members. But I think the fact that because of how, you know, I don't think a lot of C plus plus devs in particular understand that rust doesn't have type based aliasing analysis and only has the ability based. And so I think it's very tricky when you're sort of like using the unsafe cell, converting it to its underlying raw pointers and or deriving the mutable pointer from it. When it when it's really permissible to just go ahead and turn it into a hard mutable reference.

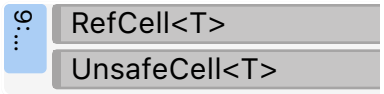


Participant
And I don't just mean the sense then then like, because you know how RefCell is, it's like ref counted. So it makes sure you don't create more than one concurrent borrow. I don't really think that's too hard in practice. Because you know, if you primarily work through the UnsafeCell directly, if you pass around references to that, it's a lot easy. It's very easy to just, okay, I only have one mutable ref here. I'm just going to keep it in the function scope. I'm not going to let it escape. It's pretty easy to like not double borrow, but it's hard to know when it's kind of safe to be like derive a mutable reference from it just in general.



Interviewer
Gotcha. And is there a reason I guess why you aren't using RefCell in these cases? And that you're just going with a pure UnsafeCell?

Participant
You know, that's a good, I think it was just like, I wanted that a hella sick perf, bro.



Interviewer
Gotcha. Yeah. Hey, yeah, totally get that.

Participant
Because I was like, you know, I've been doing C++ for so long that I'm like, can I handle actually you know, keeping a mutable reference in scope and not letting it escape. And I was like, yeah, I can do that. Plus I did, I did do some prototype testing on the Rust playground. I just like copy pasted one little function. I was like, okay, I have like an Rc UnsafeCell. Let's see if, you know, a rough gist of what my code is doing is valid in Miri in the past. So I was like, all right, we're going to spam the server without the code base then.

9:13 I just lik...
Arc<T>/Rc<T>
Running tests through
UnsafeCell<T>

Interviewer
Gotcha. So first you're confident in your ability to reason about lifetimes of mutable references and you want that extra performance. And then also in addition to that level of confidence and your motivation, you tested things out in Miri. And were there anything, any of the things you did to like, did you do any performance analysis of [unknown]?

Participant
Okay, gotcha. Gotcha. Yeah, I probably could use reps. I think I also just did it because I don't know, maybe like the ergonomics of it, because I was like, I'm primarily working mostly with raw pointers when it comes down to it. Because a lot of the stuff is like the C APIs and I'm leaking and I'm unleaking things. So I was like, I could use a RefCell, but it's a lot of ceremony for basic because I was like, you know, the safety RefCell would have added is actually pretty minuscule compared to like, just the sheer complexities of like, am I actually associating the right pointer and the right structures and that getting through in the right way and stuff like that. Yeah. Yeah. So I was like, you know, RefCell does add, Yeah, RefCell does add some extra safety. But when it comes like to the big, broad picture of things, I'm just going to be everything's going to be caught by Valgrind, regardless.

9:14 I think I also just did it...
Easier or More Ergono
RefCell<T>
9:15 Yea...
Dynamic Analysis S
RefCell<T>
Valgrind

Interviewer
Gotcha. Okay. So you're, you have that like, sort of safety net at the end of the day with Valgrind.

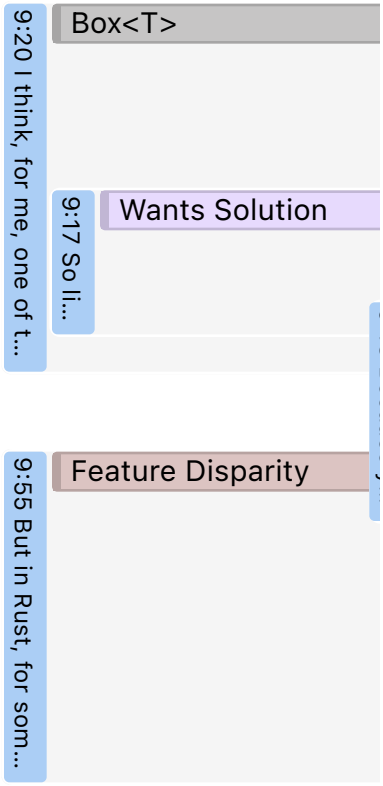
Participant
Yeah. Yeah, I'm like, I could be using RefCell, but it's a lot of work, or it's a lot of extra ceremony for what's so unsafe. And I'm like, so as if I'm not using Valgrind, then I'm iterating through all the sanitizers. Typically, I run everything, if I can help it, like, unless there's a compelling reason not to, you know, it's like, you got to use MemorySanitizer, the AddressSanitizer, the UB one, and then Valgrind. Those are pretty good together.

9:16 And I'm...
Dynamic Analysis Safe

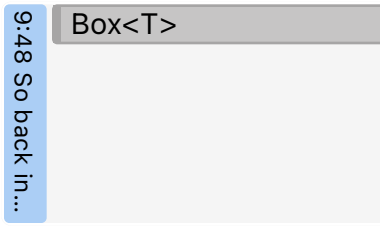
Interviewer
Gotcha. Okay. Yeah. So you, with that combination, that lets you feel more confident in your reasoning about these things because at the end of the day, you're going to catch it with one of those tools. That was my

next question, because you mentioned you're leaking and unleaking a lot of things. I know that that can be an issue with Box and Rc, where once you unwrap that, you need to make sure that you wrap it again to avoid a memory leak. That in particular, been a challenge or like, have there been other specific issues related to Box and Rc that you've run into with unsafe?

Participant
That's actually been pretty easy. I think, for me, one of the very subtle things they don't tell you about unsafe rust is, for some reason, they really overengineer Box. So I see all these kids on Discord. I call them kids. It's because I'm [age] and they're like maybe [age] or something. So to me, I'm just like, yeah, I see them talk about how under these like Box rules, it's illegal to, what was it, treat Box like for pointer stability. So like, if you have a Box and you get the raw pointer to the underlying object, if you relocate the Box, it invalidates accesses to those old pointers. And I was like, well, that's dumb. Because you know, in C++, that's like the bread and butter of all the IO code you'll ever write in your entire life. Like you have a unique pointer in C++, you give the raw pointer of to the interface, and then you relocate the unique pointer, but you still rely on that pointer having valid access tags. But in Rust, for some reason, they took Box and they were like, we're not about that life. So I think in that sense, it's very subtle. And I don't think many C++ developers would ever expect that kind of restriction. Because it's just insane, because it's what we do all day every day, you know, we're like, hey, we have, we need pointer stability, we're going to relocate, or you know, we're going to move the unique pointers we need to. And we expect that pointer we originally gave it to stay valid forever, or at least for the lifetime of the unique pointer, really.

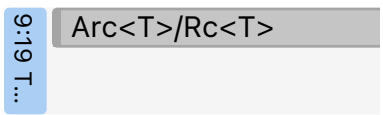


Participant
Yeah, that was like how literally all my azio code was structured, because in azio, it's primarily callback based. I mean, now, you know, C++ has co-routines now, of course. But you know, it didn't always. So back in the day, you know, you would have your callbacks that store a unique pointer to all your data. And so you would be moving that guy around everywhere. And you really do expect the pointer stability to come into play. So yeah, with Box is a huge foot gun. And in general, I really don't like Box. And I don't really use it.



Interviewer
Okay.

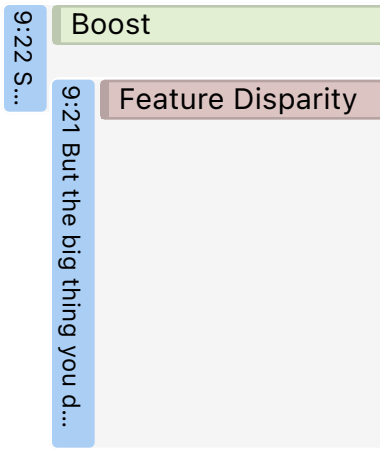
Participant
Typically, I either am using a vector, because I usually need more than one object at a time. Or I'm using something like Rc. So, you know, Rc insert cell type here.



Interviewer

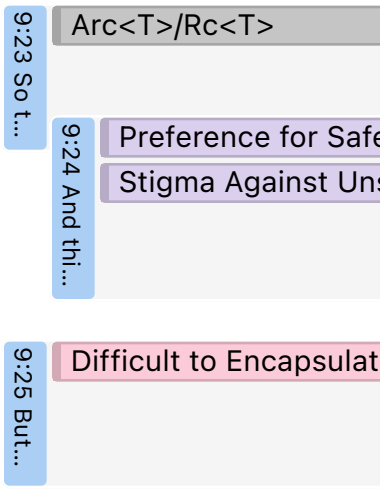
Gotcha. Okay. Yeah. So I guess with with RC, then, are there any aspects of Rust's, Rc that are different from equivalent patterns and C++ in the same way that you encountered that with with Box versus unique pointers?

Participant
You know what, I actually really enjoy Rust's shared pointer design. So in general, Boost also has a local shared pointer, which is, you know, a single threaded shared pointer type. But the big thing you don't see C++ steps culturally do is we never leak, we never leak a shared pointer in C++ in the sense that we never go because here's, first of all, with shared pointers, it's two pointers in size, in almost all cases, because you you can accept a raw pointer and participate in the ownership. So you have to have disparate allocations for the for the header where you keep in all the counts, and then for the object itself. But in Rust, you can't ever do that you you only create one allocation, whereas the frame or the header data embedded with the object. So in Rust, you see everyone leaking shared pointers everywhere. But in C++, we'd be like, you can hack it in with Boost. I've done that before, where Boost has like a little intrusive reference count thing you can inherit from. And that emulates it pretty well. But culturally, it's really not the norm. Someone would be like, you're a weirdo if you did this.



Interviewer
Gotcha. Okay. Okay. Interesting. Yeah. And then so the last one you mentioned was MaybeUninit. Could you describe the situations that you generally use that?

Participant
Oh, like the leaking? Yeah. So what was it in this little IO runtime I've written? What was it? You can attach user data to these like little IOuring-specific, like IOuring, it works in pairs of you have a submission entry and a completion entry. And so that they work in perfect pairs, you can store one point is worth of data in those little completion pairs. So typically that's what I do. I just I RC, you know, some state, and then I leak it to the IOuring runtime itself. And then I just get it back and then I unleak it and everything works great. And this really has to be done because I would actually argue it's harder to actually author a library in Rust than in C++. Because in Rust, there's no such, I mean, everyone wants a safe interface. You know, no one really wants to go use the unsafe heavy ones. So when you're making a sound, like they call it soundness in Rust, you know, and normally it's supposed to be saying it just doesn't happen to be. But in Rust, what you do is you have to make a sound, it's very hard to make a sound interface around an unsafe thing in Rust, just because if you give up like a handle, like a little ray type in Rust, you have to code around aggressive dropping and aggressive forgetting.



Participant
Like they found out that you could easily create leaks. And so they made forget and drop safe functions. So you have to be prepared for a user just

randomly dropping and randomly forgetting whatever you hand them. That could be a giant pain in the ass. Because normally in C++, we're like, Hey, we're going to give you a type, its destructor runs, you know, like important code, we need that to run. So if you don't run it, that's UB. But in Rust, you can't rely on that because drop and forget are safe.

9:26 Bec... C++ is Easier to Use

Participant
So if someone forgets your handle, you have to code for that. If someone eagerly drops your handle, you have to prepare for that too. And so when you combine all these intersecting things, it's just, oh, the burden is such a pain in the butt. Like the vectors for the iterators for vector are very hard to write to be drop safe. Like I had, when I was implementing vector, that was arguably the most annoying part of the entire implementation was the iterators, especially drain. Like drain was the worst. I hated that one so badly. Or maybe it was the splicing one, all the iterators were awful to implement. And that's just because they have to be handled, you know, like their drops in the middle of iteration, and it's just all the stability and all the panic safety, you have to add two. And the worst part is the rules around double drop or like double panic. So like if you panic in a drop handler, what happens then, a lot of that stuff isn't nailed down.

9:27 I hated that o... Difficult to Encapsulat

9:56... Shifting Ground

Participant
You know, I was kind of disappointed because C++, you know, I think historically we learned if you throw from a destructor and C++, it really should just terminate the program. So I was hoping Rust would have learned from that and been like, Hey, if you panic in a drop, we're going to terminate your program. Yeah. I don't know if they've ironed out those rules, but trying to prepare for that kind of stuff is just insane.

9:28 So I... Shifting Ground

Interviewer
Gotcha. Okay. Wow.

Participant
So in some ways, the undefined nature of Rust really, as a library author, it makes you very confused. Like, what should I do? Like, what if I am handling up, you know, what if I am handling up and unwinded my drop? And I panic again, you really don't know what's going to happen. Yeah. Yeah. So it's both difficult because you're inherently reasoning reasoning about like a complex memory safety scenario. But you have sort of these two other factors, which is there are safe operations that can invalidate what you're currently iterating on or just operating on. So you need to accommodate for that to happen. And then also the rules about things like drop order or handling of panics are fuzzy. So even if you are attempting to do things safe, it's like there isn't a clear set of guidelines. Is that an accurate kind of something? Yeah. Yeah. So real guidelines for library authors would be very appreciated.

9:57 So i... Shifting Ground

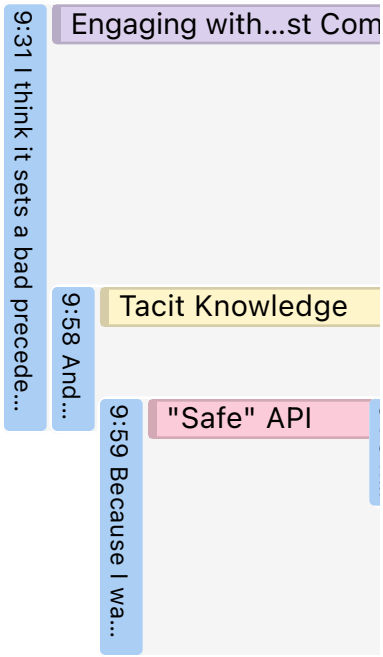
9:2... Unsafe is Diffi...to Und

9:30 And th... Shifting Ground

Interviewer
Gotcha. Gotcha. Yeah.

72

73 **Participant**
So in some ways, having a standard to fall back on for behavior is really nice. Like, I could just pull up the C++ draft and be like, you know, once I sift through the wording long enough, you can kind of figure it out. Yeah, sure. But yeah, overall, I mean, I think as a system, it still works pretty well, though. I've um, it's really funny. I get very triggered nowadays by people who call Rust a memory safe language. Because now that I've kind of put in some of the work, I'm like, I'm like, okay, it's easier to write memory safe code. But to call Rust memory safe is just like flagrantly, I think it's dangerous. I think it sets a bad precedent for the culture because I remember very distinctly once I was in [forum]. And I was asking, you know, I was still like, I'm still really learning Rust. I've only been doing it for maybe like a year or two. And so I was asking some fellow [users], I was like, you know, do you run your production tests through Valgrind?

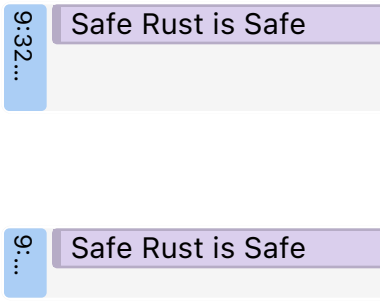


74

75 **Participant**
And they were like, why would I use Valgrind? And they were like, I'm using Rust, I don't need Valgrind. And I was like, after having implemented Vector, I was like screaming in my head, I was like, Oh my God, you need Valgrind so badly. Because I was like, no library author knows when they're going to trigger UV. Like, you know, we do our best to give you like a sound interface, but God knows if there's a hole in it. So it's one of those things where it's like as a library author, you could try your best, you know, you write tests, you write maybe some fuzzing tests, you get all that stuff going, but you never know when a user is going to stumble like into something you just didn't cover.

76

77 **Participant**
And so I was like, Oh my God, this culture is dangerous, because you have people who like genuinely believe, because I know it's like, if you're using safe code, your code is supposed to be automatically safe. And I'm like, well, that's true. And it's also not true in the sense that like running Valgrind is basically free. Or even using sanitizers, like running sanitizers and Valgrind are basically for like all you do is you just go, you know, Rust C dash Z sanitize equals, you know, whatever. But the culture was like, no, I'm using Rust, it's safe. I don't need to do all these other things. And internally, I'm just screaming because like before I ship anything, you know, I'm running through like four different sanitizers and like any tool I can get my hands on. Yeah. Yeah, she is.



78

79 **Interviewer**
So is I guess the one last type was MaybeUninit. I know that's like, I've seen that used in initialization where it would happen beneath, like a safe wrapper or other cases were that like what cases would you generally use that for?

80

81 **Participant**
That is literally one of my favorite abstractions in Rust. And I'm actually

happy to bring it up because it's such a cool intersection of features that how the MaybeUninit type in Rust works is it's a transparent union of the type you're, you know, templating on and as DST/ZST. And so the way, so it's a really cool intersection because Rust has transparent representation where the API of the struct is the API of the underlying type. Like I don't think enough C++ devs appreciate that. Because it means you can actually have an array of these types and you can treat it as an array of the underlying type. I mean, you can't do that in C++ because when you have a struct, you have the possibility for tail padding and the APIs are different. So they don't really, I don't think they get jazzed enough about that. And then also you make it non-UV/UB because you make the active member of the union to ZST. And so when I first saw that, my mind was blown with just the sheer brilliance of like intersection of like language features and library design. I was just so that's definitely my favorite type.

9:33 Like I don'...
Feature Disparity
MaybeUninit<T>

Participant
And primarily I really only use them for implementing vector in the general case, which is super cool because Rust even puts that in their interface, like you can get the uninitialized part of a vector as like a slice of the maybe on in it. And I was just like, God, dude, if we had something like this in C++, I mean, you really can't do it in C++ because the complexities they put around containers really prohibits it. So I was very proud of Rust's library team for coming up with that.

9:61 And...
Feature Disparity

Interviewer
Gotcha. So it's like a perfect interest. It's like a very well-fitted solution to problems that you've encountered with C++.

Participant
Yeah, I think the ability to represent uninitialized storage via the type system and how they have it work so well is what just makes it so beautifully brilliant. Yeah, because that's the thing. I think maybe you could try to hack something in similar to C++, like in maybe something similar to C++, but it would be a strict downgrade compared to Rust.

9:34 e...
Fanboyish

Interviewer
Okay. I guess in framing this, I see like you think about this for a moment. So you have your use for like when you're talking about RefCell in relation to UnsafeCell in general, in general, it was like, okay, you're confident in the underlying principles from C++ that help you reason about the use of those types so you can just use unsafe cell because you want the extra performance. But in this case, it's almost kind of like a reverse direction type thing where there's an existing problem C++, the fact that you can't easily represent uninitialized memory and then MaybeUninit just provides you a really elegant way of doing that. I guess would it like, are there problems or challenges that you face in using MaybeUninit or is it more the solution to challenges that you've had in other languages?

81
91 **Participant**

It's more of a solution than anything. Like using it for me at least is that simple, which is why I love it so much because it's just such a such an elegant and simple solution to a very tricky complex kind of problem. And it's also really the only way you can get piecewise construction of objects in Rust to work really, as you have to do it through the MaybeUninit. Which is one of those things I'm kind of disappointed on, like Rust doesn't have an analog for placement new. Like if you wanted to construct an object directly in some storage, you have to do some weird offset of pointer read write voodoo or you just mem copy an existing type in, which I mean, it's okay, I guess.

9:62 ... MaybeUninit<T>

9:63 Which is o... Feature Disparity

Interviewer
Gotcha. Gotcha. So I guess moving past specific type usage, you, let me just double check this quick. So you mentioned doing some FFI stuff. And the languages that you are calling your, so you're only calling things bi-directionally, it seems like into, sorry, unidirectionally. So you're calling a C and C++ from Rust. And then he said that you generally write bindings manually without using any other like tools to assist you. Could you talk about your, oh, sorry, said again?

Participant
I don't trust them.

Interviewer
You don't trust them?

Participant
Okay, yeah. I don't trust bindgen at all. I wouldn't even. Also, I'm not wrapping that much FFI.

9:3... Generation VS Validati

Interviewer
Gotcha.

Participant
Well, actually I am. But I don't like those automated tools.

Interviewer
Gotcha. So yeah, I guess describe your experience writing bindings manually and why you choose that over writing, or over the automated tools.

Participant
Well, I have broken down in my old age. I now use libc, which has been a huge quality of life upgrade because having to work with ErrorNo, their approach to it was a lot better than mine, which is very ad hoc. And yeah, so I finally use libc. Someone showed me the Nix crate, which that was pretty cool. But in general, I don't really like, I don't like the generated

... Generation VS Validati

code phase of build systems. Like if I can avoid it in general, I'm going to avoid code generation. I know Rust makes it like, you know, somewhat approachable with the build.rs file. But let me close that door a little bit.

9:64... Generation VS Validati

Interviewer
No worries.

Participant
Yeah. So like build.rs, like bindgen's okay. I like the dedicated crates a lot because it comes with like a lot of the documentation and all that kind of stuff. Writing stuff manually by hand isn't really that bad. Because I mean, all you're really doing is just copy pasting the declaration and then you switch the types out. I mean, I very sloppily always just use i32 instead of int. I'm like, it's always going to be 32 bits, guys. Show me one platform. I mean, someone probably could, but I don't expect my code to ever go there.

9:36 Writing st... Generation VS Validati

Interviewer
Brb, inventing a 54-bit integer just to mess with you.

Participant
Yeah, I was going to say, like, you know, those C rules around integers were arcane. I mean, for the time it was very relevant because they were like, man, computers, we're going to figure these things out one day. But you know, now it's like 50 years later. And yeah, we have like the dominant architectures and everything's kind of like almost set in stone, basically. But yeah, so I primarily call C functions from Rust. I don't go the other way. Like theoretically, if I was working on a C++ codebase, like at my last job, I would have, you know, made Rust. I would have, I would have done it, you know, like take a Rust lib, make all your extern C functions, and then just call them from C++. But that's a lot of work. It has to come with a lot of benefits, make it worth it too. Because once you introduce the, so that's the thing, you know, when you start intermixing C++ and Rust, it gets a lot trickier just because of how you want to have those two ABIs communicate with each other. Like in general, I prefer only using the system ABI, you know, like C ABI. And so when you do that, it gets kind of, I don't know, it gets kind of annoying. So I typically only try to go one way, which is consuming C from Rust directly.

9:37 when you... Different FFI Memory I
Simple FFI

Interviewer
Gotcha. That relates pretty directly to my next question then, where I guess both in terms of the ABI as well as the memory models of each environment, what are some of the challenges that you've faced in reconciling the differences between Rust and C and C++ when you're calling those foreign functions?

Participant
You know, I kind of just YOLO call them and wait for Valgrind to tell me if I did it wrong. Gotcha. I mean, specifically, I'm like so well trained as like a

9:3... Dynamic Analysis Safe
Valgrind

C++ dev that I usually always send initialized memory or stuff like that. So it's not really too different. I'll actually, you know, I'll actually say there is one, there is one massive difference though. So in general, binding, binding isn't so bad. The real problem occurs when C libraries use things like macros very heavily. So I found that to be really problematic. So what was it? The IOuring author, he has a helper library called liburing. So most of that library's interface was living in the headers. They were all like static inline functions. So they were permitted to be, you know, redefined in every translation unit. But the problem is, so that was one of those things where I was like, I was glad I wasn't using bindgen, because I'm like, how would bindgen respond to a function that doesn't exist in terms of the compiled library? You know, like would it just reimplement the static functions from the header? Would it have just like skipped them? Like what would it have done? So what I wound up actually doing was I made a pull request for liburing to update its make file to also recompile, like to use a macro to remove the static inline. So that way you actually got a compiled library out of it.

9:... Tacit Knowledge

9:66 But the pr... bindgen

Limitation of Binding T

Interviewer
Okay, gotcha.

Participant
It was actually, using Rust, It was actually easier to make a pull request to the library I was using and get it to actually build the binary artifacts I needed. Instead of just like trying to make the bindings work by hand or something like that.

Interviewer
Gotcha. Okay. Huh. So it's mostly been problems related to macros. And your experience in C++ has given you practices like never passing uninitialized memory where you feel fairly confident. Oh, and I was gonna say, and how do you, and how do you use Ray to really make using C interfaces easily?

Participant
Because like a lot of C interfaces, they really just give you, here's the init function, here's the cleanup function. And basically that's trivial, just wrap with like a drop type, you know. Okay. So I mean, that kind of, once you get that pattern down, it's really, really easy. And it's kind of funny because I'll see a lot of fledgling Rust developers who aren't used to that technique yet, and they complain a lot about how hard panic safety is. And I like, I really think is just use drop guys. That's what is there for drop is your friend. Everything needs to be, because you even see that pattern in the stdlib too, like they use an internal type called drop card a lot. Like they'll make a, they'll make a like an ample of something. They'll have a little function defined struct with a, that's another thing I love about Rust. You can have a, an entire structure defined inside the function with its drop info attached. I would say that was kind of cute. But yeah, so it's, there's C libraries that do things like, they'll use long jump or set jump. That's a problem because Rust can't handle that. Yeah. So you have to

9:39 Bec... Simple FFI

9:67... Different FFI Memory I

create an intermediate C function that wraps all that jumping for you. And then you can call that from Rust.

9:... Different FFI Memory I

Participant
Gotcha.

Interviewer
So the example here is libpng, actually. And here's the worst part. The PNG crates in Rust aren't really as good as libPNG. And I know the PNG gets a lot of flak for having all those like memory safety bugs. It's really funny to bring it up because it's like, that is like one of the example ours of why like Rust is supposed to be so much better, but only one that has all the features. So like the PNG crate, I tried using it in the past. And the problem was it has like a tenth of the features. And I mean, I get it because it's like, there's a lot to code and it's a volunteer effort. So it's whatever the maintainer is willing to put in. I mean, yeah, it can, it can do full reads and writes of PNGs with like an alpha channel and stuff, but it's not going to do like half the features that live PNG actually has. And I was like, well, that's lame because it's kind of a downgrade, but then you use the PNG and then it has like long jump in it. And you're just like, uh, it's so hard.

9:68 So like the PNG crat... Rust Rewrite is Incomp

Interviewer
So speaking of difficult programming patterns, this next question is pretty broad again. Describe a bug that you face that involved unsafe Rust and sort of, if you can think of a particular example, go into how it manifested and then what the, what the fix was.

Participant
Oh, man. This is funny. This is actually from liburing, the bug I encountered. So the problem was the library itself gives you quasi uninitialized data. So I was getting these completion entries in from, you know, the IOuring runtime itself. So I was getting these completion cues in and you examine some part of the data structure is, you know, a
127 member called user data and user data is like that eight by things. You normally store a pointer in it. So the problem was I, I actually didn't set the user data when I made the submission. So when I got the user data from the corresponding completion, it contained a random pointer. The problem was liburing was giving these random pointers. They were real pointers. They actually pointed to these leaked RCs I had given in the past. So I had this horrible bug where like random co-routines weren't being resumed properly or they were being double resumed just because of how it came in all, how it came in all jumbled from IOuring, you know, completion submission cycle. So that was pretty interesting to debug. And those ones, those ones were like valgrind didn't catch it because technically like the data was set. It was just, it was given random bad values. And it was really funny because I saw the Azio author, he actually had run into the exact same bug I did. I saw the commit where he talked about it and I was like, that hit me too. I was like, oh my God, that's so funny.

9:40 So the problem was I, I actually didn't s... Arc<T>/Rc<T>
Temporal Memory Bug
Uninitialized memory
Using Uninitialized Me

9:69 And... Missed a Bug

136

137 **Interviewer**
So now I learned set the user data to null manually, set it to null and then it'll come back correctly. Because I was relying on that. I was like, okay, I'm going to get a completion entry. The pointer is going to be null. This code is going to work. Sweet guys, check this out. Then I had all these really obscure bugs where I was like, you know, it's really hard testing an IO runtime. So I think I caught it because I had like, I had perfectly arranged the like the house of cards where I had like three timers. They were like, two of them were doing weights. One of them was just starting and that was when the bug manifested. And those are always the worst because they're basically like heisenbugs where you're like, but all my other tests passed. I swear.

138

139 **Participant**
So yeah, I finally found it. I was like, oh God, that was awful. That was, it was a lot of print line. It was a lot of printing everything. And I had to like get there with my computer screen all hunched out like many tracking pointers being like, I leaked this pointer to this coroutine. You know, I call them coroutines. That's why I call the rust a single way. I know some people get kind of like persnickety, but they're coroutines. So I had this thing where I was like, yeah, I was matching all these pointers to the coroutines. And you know, I finally found it because I was like, wait a minute, that pointer shouldn't be there. I was like, look at the screen. I was like, yeah, that pointer already associated with complete coroutine, you know, we should be good. So that was how I found it. That was, that was interesting.

9:... Printf-Style Debuggin

9:42 I leaked this poin...

140

141 **Interviewer**
So then I guess going into more bug finding then you selected every single bug finding tool that I listed. And from earlier, you mentioned your approach is like, because you have these tools, it's like, yeah, throw all of them in testing to catch everything. So I guess, could you describe your experience with these? I guess both like all the using the sanitizers, maybe in particular, like if you if you use Miri, hearing about how that has been for you.

142

143 **Participant**
Oh, yeah, I've learned historically, sanitizers need to be run separately. So one problem with rest is so was a little bit about me. I work for [organization]. So we're basically paid staff to maintain [redacted] libraries. So a big part of [library] is, you know, we have like one of the biggest test matrices around. So one of the things is we have we have our own internal build called [build]. So with [build], you can have it run like, like a product matrix of tests very easily. So if I want to write something in [build] to run, like, I'm like, okay, [build], I want you to run my tests. But I want you to do four permutations where you run the test with just address sanitizer, then just the undefined sanitizer. And then I'm doing some concurrent stuff. So just the thread sanitizer. So [build] has a way of doing that in one command where it's like, okay, sweet. So I'm

running all your tests for, you know, ASAN, UBSAN, TSAN.

Participant
But in Rust, that doesn't really exist. Like you can't tell cargo, hey, cargo, run my tests and run it as like a product matrix of all these configurations. You know, you basically have to like, I'm going to write a shell script. And then I'm going to have it vote cargo with, you know, dash C for this one and cargo dash C for this one, you know, so on and so forth. So it's definitely a strict downgrade. So that that experience is kind of lame. I found out you can set a little runner environment like in a cargo dot ENV or whatever for your workspace, you can get you can have an outline through that, which is pretty convenient. I'm not really a huge fan of Rust's fetish for like environment variables. But if you're setting environment variables in a config file, it's not really a big deal. You know, because like the cargo, the cargo configure whatever has like a dot ENV section where you can set everything. And I'm like, ah, that's okay. But I'm like, it's not as good as CMake's toolchain file.

9:43 But in Rus... Wants Solution

Interviewer
Gotcha.

Participant
Which is where I'm coming from. I'm like, man, I'm like, if I could just call Rust from CMake, that would be the dream for me, which is I know there's a lot of like young rustaceans who would be like freaking out at that. But haven't you CMake for so long and done so much? Like, a lot of people don't even realize you can use CMake to easily package up a Debian file. You know, like my last company, I had our entire application running as a dot DEB. And it was glorious. So yeah, if CMake had first class Rust support, oh, that'd be beautiful. Cargo is okay. But I'm like, I kind of miss the days of manually controlling, I'm gonna have a static archive here and another one there and I'm gonna link them for just as executable. Like I really miss that kind of control CMake gives you. Whereas cargo is a lot more declarative, you know, it's like that. It's the convention over configuration school of thought, because I know rails is very similar where it's like, you put your files in this folder and they're automatically part of the whatever. And I kind of hate that.

9:7... Wants Solution

143

151 **Interviewer**
Gotcha. Yeah. And with you. So you have the sanitizers then. What about Miri? How is has this Miri been useful for you?

152

153 **Participant**
Yeah, yeah, I use it was I test my vector through it regularly. Technically, I've actually had I think less. So what was it to test my vector implementation, I copy pasted the stdlib's test suite for it. That was how I learned the stdlib has all of the black magic that they get to expose to it as stable to users that I can't, which I think is very unfair as a library author. I feel kind of like, I feel like the language designers or, you know,

whoever the powers that be are like almost purposefully giving my powers a library author, because I think it's unfair that they can use min stabilization, and then expose that to stable tool chains. But if I want to benefit from all the specialization, I have to be nightly. And I'm like, Oh, well, that's not great. Because, you know, it's not like I can't use the features. It's just, I don't want users to have to use nightly tool chains, you know, like, I think live stable is a pretty solid, you know, requirement. It's like, Hey, you know, just run Rust update or whatever we know again, you can keep using the live.

Participant
But yeah, so Miri's been really great for that. Especially because it was like a vector, you're doing so much weird stuff to it because the iterators, like when you start draining and leaving holes in it. That's part of the problem why it gets so complicated. Because if you like the drain iterator removes a sub slice from the vector. So that's when it gets tricky, because it's like, you know, you have to like set the length of the vector, you have to like truncate the vector essentially, leave the elements behind in the trail. And yeah, so there Miri's pretty clutch, as you know, I'm not doing any ffi. And it's very important that you get a foundational container correct. So you know, it's funny, technically for work, I work on boost hash tables, not the algorithm stuff, I don't want to take credit for that. It's my brilliant coworker who came up with all the algorithms. But I've been thinking about translating his work to rust. And then something like that, Miri would be absolutely paramount.

9:7… Miri doesn't s…ort this

9:44… Running tests through

Interviewer
Okay. Are there any particular patterns of bugs or like cases of undefined behavior that Miri has been helpful for you in detecting?

Participant
Yeah, I'm trying to think as I know I must have had some Miri. I found Miri kind of late. In the sense that I had already written almost all the vector by the time I found it, and I was using valgrind the whole time. So it's not like Miri showed up and was like, here's all your bugs. It was, you know what it really is, here's a massive load gun. It's getting an address is what's weird about Miri. Then by which I mean, if you want to get the address of an object, you really shouldn't form a reference to it. So in Rust, they're called places. When you want to get the location of a place, you have to use the macros, you have to use the adder of or the adder of mut to get that properly. And I remember there's another C++ developer I know from the C++ Slack who started trying out Rust, and they wanted to cache a raw pointer for writing later on.

153

161 **Participant**
And their code looked really simple, looked very simple. They were like designing a computation graph. So, you know, it was like interlinked node, you have like inputs, outputs all linked together via separate structures. So they were trying to get the address of a variable. And they did it by binding a const reference, and then they like as casted it to a

9:… Borrowing or Provenar

Stacked/Tree…rrows Vi

pointer, very standard stuff, right? Well, fun fact, that's wrong. Because when you cast that pointer, you invalidate the borrow stack. So Miri started screaming at them, like, hey, you're generating a store to this location, but it doesn't have write tags. So stop doing that. Everyone was like, everyone was trying to figure out, well, why is this code wrong? Because we all looked at it, and we were like, well, have you used UnsafeCell? Because we thought, oh, it's complaining about a lack of a write tag. Maybe you just need to tell it, hey, this is interior immutability, you can write to it, go nuts, but the error is still there. And it turns out it was because they were doing that stupid reference binding. And so the fix that someone else caught was just use addrof.

9:45 So t...
Borrowing or Provenar
Stacked/Tree...rrows Vi
9:46 Everyone was lik...
UnsafeCell<T>

Interviewer
Okay, interesting.

Participant
It was funny. That was a clairvoyant moment for me because I was like, oh my God, in C++, you have to call address of because some individual may overload the little ampersand operator. But in Rust, you have to call addrof because a reference invalidates your borrow stack. And I was like, oh my God, they're the same language, guys, you have to call address of with both.

9:72 That was a cl...
Shared Experiences

Interviewer
Gotcha. Oh, this is interesting. Different reasons, but same practice.

Participant
Yeah, totally funny. And I was like, that's a really easy. They're still kind of complaining about it. But I'm like, it's just like C++, just call address of. You don't have to understand it. You just call the function, you know, you need to call and, you know, move on.

Interviewer
Gotcha.

Participant
So that was one of those things. I think I had bugs in my vector about that where I was binding references to get addresses and Miri was like, I can't do that.

Interviewer
Interesting.

Participant
Yeah, I haven't run into that problem myself using Miri, but I'm glad to know that it exists so that I don't have to. Yeah. Because I know that I'm gonna, that that footgun is gonna show up eventually for me, I'm sure.

178

179 **Interviewer**
So yeah. So one final like question, I guess that covers pretty much everything I think we talked about today, which is, have you found issues and unsafe common problems, challenges you have in writing unsafe code or using it correctly, that you feel your bug finding tools cannot help you with, or that you wish there was a particular tool that existed that could do something for you?

180

181 **Participant**
Oh man. Someone linked me to this great. So basically, my ideal for Rust would be Valgrind and Miri having a baby. Because I love Valgrind because it tracks allocations. So basically, the problem is sanitizers are kind of weak. Sanitizers are very, very fast. Like the instrumented binaries, they pretty much run at like full speed. I know there's like a game dev out there who's like, no, they don't. And I'm like, well, of course, because it's instrumented, bro. But they're really fast. Valgrind is mega slow. If I was writing a GUI, I probably wouldn't be using it. I don't know how realistically you would be able to like run a GUI through Valgrind. But it tracks file descriptors, it tracks uninitialized reads of stack locals, and it tracks all that great stuff. But the problem is it doesn't have all of that stacked borrow checking, or I guess the youth talk about the tree borrowing. There's a new borrowing every week, you know, whatever Miri runs, Miri runs, I guess. So the problem is you don't have all that. So there was someone who was telling me about this project called crab cake, where I guess it's someone who's actually trying to do that. He's trying to, at least I can get the, I might, I might just email you the link, but it was called a crab cake. Yeah.

9:73 There's a...

Krabcake

182

183 **Interviewer**
Oh, I had really, if you can find a link to that, that would be amazing.

184

185 **Participant**
Yeah. You know what, if you're not on the Rust Community Discord, oh, wait. Well, I saw your, I saw your study link there, which is how I found it.

186

187 **Interviewer**
Oh, gotcha. Okay. I know I'm just like Ian McCormack over in the Rust Community Discord. Or just feel free to like send me a link here in Zoom or just like an email whenever you get the chance to be awesome if you can send that my way.

188

189 **Participant**
I think I have it right here. How do I text in Zoom?

190

191 **Interviewer**
Oh, it should be like there's just a chat window at the bottom.

192

193 **Participant**
Oh, there's a chat. Boom. All right, that should be the, that should be the crab cake link.

194

195 **Interviewer**
Awesome. Perfect.

196

197 **Participant**
Oh, I got you. Yeah. Someone came up with crab cake, and it looks like it's just that it's like Valgrind plus Miri stacked borrow checking. And I'm just like, Oh my God, this is everything I've ever needed because it does have, here's the thing. I actually do real networking tests in my Rust code. Like, I actually call out to Google.com and I do a TLS handshake with it. And you know, I actually read the response for getting the homepage. So in some ways, I like tests that really test. So Valgrind is like a necessity or at least or sanitizers, you know, like I have to have the actual runtime, like, like we said, guys, let's literally call it to Google. Let's actually call the DNS server. Let's actually like do the real stuff because, you know, users aren't going to be running hermetic little local host or using mocks, you know, like you actually need real world testing. So yeah, that would be the biggest. Oh, also in the ability to run builds as a product matrix, like I want to be able to test, you know, like give it a configuration. Here's the four sanitizer, like here's the three sanitizers you need to use. You know, when I run, when I run cargo test, it'd be nice if you could run, you know, my test suite would just you be saying and then just MSAN and and then just ASAN.

9:74 Som... — Krabcake, Valgrind, Wants Solution

9:75 Oh, also in th... — Wants Solution

198

199 **Interviewer**
Gotcha. Gotcha.

200

201 **Participant**
Yeah, that would be so freaking amazing.

202

203 **Interviewer**
Oh, yeah. All right. Yeah, no, I, yeah, those, those are two, yeah, I can totally see how each one of those fits into a particular like niche and the development that you've been talking about. Yeah. And that so yeah, if you can find the link to that first one, I know I'm just like Ian McCormack over in the Rust community discord or just feel free to like send me a link here in zoom or just like in or just like an email whenever you get the chance to be awesome if you can send that my way.

204

205 **Participant**
I think I have it right here. How do I text in zoom?

206

207 **Interviewer**

Oh, it should be like there's just chat window at the bottom.

Participant
Oh, there's a chat. Boom. All right. That should be the, that should be the crab cake link.

Interviewer
Awesome. Perfect.

Participant
Oh, I got you. It's actually a legit it's a legit Valgrind fork. There's actually Valgrind in there. And there's actually commits too. Like the guy's actually working on it. I was like freaking out. Oh, hold on a second.

Interviewer
No, no, you're totally good.

Participant
That's something. Oh, sorry about that. Oh, I was going to say I like viewing unsafe and Rust a lot differently. I feel than the majority of the rustaceans I talk with, which is more or less it documents which functions have preconditions and which ones don't. I don't really like the idea of looking at it as a marker of this function can have you be versus this function doesn't have you be because safe functions very easily have you be. I just like looking at it as like this is a function where maybe for efficiency's sake, we have assumptions you have to upholds the caller, you know, and I know it sounds kind of self explanatory, but I guess culturally it's really just like unsafe means unsafe and not unsafe. It's totally safe. So which is why I'm, you know, that's why I'm so perturbed when I hear when I hear the youth say, Oh, I don't need Valgrind or I don't need sanitizers. We need those tools as the author of an IO runtime. I can assure everyone no one knows what they're doing. We're all just trying our best.

9:47 but I guess cu...

Safe Rust is Safe

Interviewer
Well, anyway, I think that covers pretty much everything and more that I could have possibly asked. So thanks a ton for your time today. I really appreciate it.

Participant
Yeah, I have a lot of thoughts on rust. It's really funny being like a citizen of two, two worlds, because like, I love C++ so much, and I love rust so much, you don't really find people who are like big on both a lot.

Interviewer
Gotcha.

225 **Participant**
So I have a lot of thoughts ...

226

227 **Participant**
Yeah, no problem. I like, I like what you're doing because I really do believe in rust a lot. And so it's good to see people trying to make the unsafe ecosystem a lot better. Because right now, you know, I've come to view rust as like a sort of like a two tiered language. In rust, you're one of two people you can either be trusted with unsafe or you're the forbid unsafe kind of person. So I think for people who actually write unsafe code, there's just so much work that needs to be done. Like the tooling is okay, you know, like you have sanitizers and whatnot, but it's just without a formally defined spec, it's, you're really just flying by the suit of your pants and more often than not, I'm really just like, well, whatever the stdlib does, I'm going to do too. That's because they can't be wrong. They're too standard to fail.

9:76 Because right no...
Stigma Against Unsafe
9:77 So I think for...
Shifting Ground

228

229 **Interviewer**
Yeah, no, that doesn't, yeah. Interesting. Well, anyway, yeah, it's been awesome talking to you. I think I've just covered every possible thing. So it's some great hearing your perspective. And hopefully we can, we can both make unsafe rust better for that, that tier of usage.

230

231 **Participant**
Yeah, no kidding. All right, well, no, thanks man. I really appreciate this.

232

233 **Interviewer**
Hey, really appreciate it too. And I'll follow up with it.

234

235 **Participant**
Yeah.