Interviewer
So the first question is what have been your motivations for learning Rust and for choosing to use it?

Participant
I would say I started learning Rust probably back in like 2018. And the motivation was I was really tired of writing C++. I appreciated all of the like fine tuning and the low level control that I had for the language. But it just seemed like constantly the language was evolving in a direction that made it more complicated, more confusing. There are a thousand ways to do the same thing. And when Rust showed up, it seemed like, Hey, here's a language that lets me have the low level control of C++ without a lot of, you know, generational cruft built up and with a bright future, a clean syntax and generally like nice capabilities.

3:1 But it just seem...
Cruft

Interviewer
Gotcha. So could you describe a bit more about your experience programming in each one of these languages? Like, I'd say, could you compare and contrast how you think about programming when you're writing C++ versus how you think about the same problems when you're writing in Rust?

Participant
Yeah. So for background, I worked in game development for a long time. And so the kind of C++ that we write in game development is very different from the kind of C++ that you would write if you were doing like more like enterprise development. So most of the code that I wrote was like C++ 98 effectively. We generally did not use newer features in C++, even though we did keep our compilers up to date. And like a lot of the fancy C++ features like templating and stuff, we would use very in a very limited fashion. And so the same problem would end up being like conceptualized in similar ways in C++ and Rust for like small tasks. But when you got to like larger structural problems, C++ kind of tends to like get this object oriented focus about it. You would want to like use polymorphism and virtual like make all your virtual functions and override them in your base classes in order to achieve like flexibility. And I think that was because using templates for a similar for like compile time. I'll call it compile time polymorphism, even though that's not exactly what it is. Using templates to substitute types in at compile time was just heavily discouraged. The errors that you get would be off the wall and you wouldn't know if things were broken until you actually went to use them. And it was really difficult to keep coherence through a really long chain of systems working together. In Rust, I'm a lot more comfortable using generics and treat bounds to enforce that kind of structure, even as I'm building really large interlocked systems. And so I would say that's probably the main difference for like the way that I approach structural problems. At the low level, it's not really that much different. Rust is nice because I don't have to worry about memory safety a lot. So I spend a lot less time writing no pointer checks. I can definitely say that if you've ever peeked into like Unreal Engine's code base, the number of null checks

3:2...
Shared Experiences

3:3 Using templates to s...
C++ is Difficult

3:4 R...
Memory Safety

are just insane. Even when you know for a fact that some variable will never be null, there's a null check because you got to you got to make sure.

Interviewer
Yeah. Interesting. So it seems like the way that you were describing the difference is really tied to the type system in the sense that you have this better assurance of functionality when working with Rust's trait system versus you mentioned where you're using templates. It tends to be that you don't really know if something works until you start using it. Could you give a bit more detail about that particular problem? What's an example of where something would, where you really wouldn't see that there was a problem until you used it and how Rust gives you a different experience?

Participant
Yeah. So as you might be familiar with, in C++ templates are only created when you actually use them with specific types. And so if you're relying on like Sphiné, substitution failure is not an error to build up some functionality over time, then you can get into a lot of crazy situations where you'll put some types into template arguments and suddenly you'll get some massive error that is completely incomprehensible. Talking about all of these nested template types and picking those apart is a lot of work. I guess part of that is because the compilers C++ compilers tend to generate really unfriendly errors. And the other part of it is that those only show up when you actually go to use them. And so in Rust, I'd be comfortable like writing a bunch of code and then after I'm satisfied with how it looks, I can write some tests to make sure it works. But if I want to actually make sure templated code works, I basically have to write the tests upfront before I'm even ready to like decide on a permanent structure of how I want to approach things. Otherwise, I won't run any errors that might, you know, I might get to the end of the design process and then go to do something and then realize, oh, well, this doesn't work at all.

3:5 Talking about all of thes...
C++ is Difficult

Interviewer
Gotcha. Yeah. So this is more of a transition to unsafe Rust. What has been your use for unsafe Rust generally in the applications or projects that you contribute to?

Participant
Do you mean like at a high level or like specifics?

Interviewer
Let's start with a high level and then move to specifics.

Participant
Okay. Unsafe Rust generally, I generally use it in two different ways. The first is obviously to, you know, outsmart the compiler, squeeze a little more performance out. You know, when I know that certain invariants are

3:6 Th...
Increase Performance
When to Encapsulate?

met, I can use unsafe to speed up my code a bit and then wrap that in some sort of safer, larger abstraction. And then the other side of it is I use unsafe to construct new like compiler level, well, not compiler level, but to construct new language level primitives. So like if you're familiar with Pin, Pin uses a lot of unsafe, even though it's like, this is almost a tangent, but there's this really interesting kind of dichotomy where unsafe has some really clear semantics about anything that can cause memory unsafety needs to be marked unsafe. But we also kind of co-opt it for forcing people to uphold invariants on types, even if those invariants aren't explicitly required for memory safety, if it's just like correctness or something. And so I use them in that situation as well. So like when I'm building some new language level primitives, I'll use unsafe to make sure that everybody, you know, enforces contracts by wrapping things in unsafe blocks and justifying them properly so that you can take advantage of some large scale invariants or build up some new invariants that let you write more efficient code or more simple code or more practical code.

3:6...
Increase Performance
When to Encapsulate

3:8 So like when I'...
As Documentation

Interviewer
Gotcha. Interesting. So I guess could you transition a bit more to the specifics? Like what have been some cases where you've used unsafe to increase performance?

Participant
Yeah. I'll pick out one specific one. So I write a serialization framework called [name]. And obviously a goal for it is to be extremely performant. There it basically uses a two phase system where it has to write a whole bunch of data, store some like temporary stuff, and then use all of that temporary stuff in one extra pass at the end. And so you end up needing to allocate some data occasionally. So like if you want to serialize a vector and that vector has a bunch of elements that have non-local data in it, need to store a bunch of information and then use all that to finish things up at the end. So doing that allocation normally by just using like a std::Vec or using like the built-in allocator actually ended up wasting a lot of time. And so when I was doing a point transition, I thought this is an opportunity to squeeze a little bit more performance out of it. And so I switched to using a custom like basically like a bump allocator that comes with the serializer. And it has much more restricted semantics for how you're allowed to allocate and deallocate things. And it's like extremely thorny as an API, but for the most part, if you're using it, you're writing unsafe code anyway. You understand how to like use this very like sharp edged abstraction properly. And so when I switched over to using that, it cut like, I think it was like 20 to 40% off some benchmarks. Just because the amount of like extra data that I needed to allocate was just thrashing the global allocator really hard. But using a bump allocator, it was really quick to just allocate a big chunk and then free it. And so restricting my semantics, let me squeeze a lot more performance out of it. But because they were so like particular and sharp, it had a really unsafe surface to it.

3:9...
Data Processi...& Seria

3:10 And so when I was doing a...
Increase Performance
Performance is Neces

3:50 And...
Tacit Knowledge

3:1...
Unsafe Data Struct

Interviewer

Gotcha. Um, could you describe a bit more about what you mean by sharp or thorny with this API? Like what were the challenges and using it in terms of maintaining memory safety in an unsafe context?

Participant
Yeah. So one specific one that I will mention is having panic safety is really difficult when you want to build certain APIs. So, uh, panic safety is when you're doing some operation and then you call some extra code that could panic in some way, and you need to make sure that you cleaned up what you were start, what you started doing or else you could cause memory and safety or leaks or something. And so the, uh, the allocation API that I had, uh, needed to be panic safe so that if like someone else's serialization code panicked while you were in the middle of like building up this giant list, uh, that you wouldn't accidentally go and stomp over things or use free pointers or, uh, go tread back into like data that you shouldn't be reading. Uh, so working that in was kind of difficult. Uh, that's basically an unsolved problem. Uh, we have like some approaches for fixing it that use like, uh, drop handlers, but they're kind of nasty. Uh, so panic safety is like a particularly thorny, uh, API surface. And then besides that, it's basically just it's raw pointers everywhere. Uh, it's not a really sophisticated API surface. It doesn't expose a lot of types. It's kind of kept as minimal and straightforward as possible. Uh, I could build up more API surface to make it like easier to handle all of those different components, but, uh, I guess this is also a good example. Uh, because of the way that the memory is owned, it was difficult for me to, uh, build a structure that could free that memory without using extra unnecessary data to reference a variable that I already have on hand. So I have my serializer. It knows where all the memory is allocated. But if I wanted to make like a vector that uses that, uh, it would need to hold a reference to that, uh, that allocator, even though I already have the allocator, it's in my local variable. Uh, and I ended up with duplicated like references all over the place, even though they're short lived, they only last the length of a function. Uh, and I don't really neeeeeeeed to hold on to them. And so if I make a lot of those, I end up using a lot more memory than I need to, uh, which is a little frustrating. Uh, it's not like a big deal, but, uh, I felt like that was also worth mentioning.

3:13 And so the... Panic Safety
3:12 Uh,... Shifting Ground
3:14 A... Exposing an U...e API t
3:15 Uh, because of the way that t... Difficult to Encapsulat

Interviewer
Gotcha. Interesting. I haven't heard that particular pattern. Um, so it's, it's just that you, so it's, it's not a, like a, a cyclic pattern with references. It's, it's just the fact that you need to constantly have this access to the allocator everywhere.

Participant
Yeah. Uh, and like you could like prevent having that duplicated reference in there. If you just like move the allocator into it, but that, that's not what I want to do. I want to keep them separate. I would like for them to like be able to reference each other, but I figured that it was just easier to have like an explicit, like free call and then pass the allocator back in so it could free all of its memory at the end.

Interviewer
Gotcha. Interesting. So then for that second category of unsafety that you mentioned, it was related to building primitives similar to, to Pin. Could you describe a bit more about the types of primitives that you've implemented with unsafe, like what their use cases have been, um, how you ensure that you've used unsafe in a way that isn't like breaking Rust's, stable semantics.

Participant
Yeah. Let me pull up some docs real quick. So I'm going to send you a docs link. Uh, this is for some experimental work that I've been doing. Uh, this is for like one crate in it, but you can access all the crates in the sidebar. So this is a library that works, uh, that builds up to, uh, like relative pointers, uh, in rust, which are, they have their own very particular, uh, peculiarities. Uh, but one of the main difficulties is ensuring that you have two pointers in the same allocated region in LLVM terminology, uh, which basically means that you can, uh, translate between the two pointers effectively. And so you can go from, you can take two pointers, you can find the offset between them without invoking undefined behavior. You can offset from one pointer to another, uh, and get to another place safely without invoking undefined behavior. And so one of the, uh, one of the interesting primitives that I built up was this, uh, in wrapper. It's under the [name] crate. There will be an instruct. And it's basically like a Pin, but instead of saying this item is fixed in place, it's saying this pointer that I have is located in some memory region identified with a type. Uh, and so I use unsafe there to say, take, uh, some pointer that I know is associated with some region type and glue them together and say, okay, whoever called this new function has promised that this pointer and this region type correspond with each other. Uh, and I can use that to make sure that I can take two pointers that I know are in the same memory region and do all these operations safely with them.

3:16 So this is a library th...
Pointer Arithmetic
Unsafe Data Structure

3:51 Uh, and so...
Contract or Invariant

Interviewer
Gotcha.

Participant
So that's one example. Also in that crate, uh, there's this unique trait, uh, which is related to that region. And it's, it represents a type that only one value of it can ever exist at a time. Uh, and that's used for basically like uniquely identifying those regions. So a unique type could be turned into a region just like a, it's kind of like a brand. If you've, I don't know if you've read, uh, or heard of the ghost cell paper.

3:52 Also in tha...
Contract or Invariant
Unsafe Data Structure

Interviewer
Oh yeah. I'm not too familiar on that, but I have heard of it.

Participant
Yeah. So the idea is basically you could generate a fresh type that's

never been seen before, and you can use that to like brand particular values to keep track of them to basically correspond correlate a value with a type and then use that for all sorts of like tricky operations to make sure that you remain safe.

Interviewer
Gotcha.

Participant
So I also use it for stuff like that, where I construct these new ideas that I can use to, uh, make more efficient code or make my code even possible.

3:1... Increase Performance
No Other Choice
Performance is Neces

Interviewer
Mm hmm. Gotcha. So it's, it's both like the rationale for doing this is both performance as well as just certain patterns, like maybe the idea of using a pointer to modifying a pointer and ensuring you're staying within the same region, like that those patterns are not possible without using unsafe, but you want to expose a safe interface to doing that so that you're not invoking undefined behavior.

Participant
Yeah. Yep. That's exactly right.

Interviewer
Gotcha. That's interesting. I'm going to be spending a lot of time with the docs that you just sent.

Participant
I think they are kind of exotic. So, feel free to reach out if you have any particular questions. I think the coolest one in it is probably static ref, that builds on some of the other primitives to make a zero size type that is a safe reference to a static variable scoped reference to a static variable. It's pretty neat. I like it.

3:18 I... Static Variables
Unsafe Data Structure

Interviewer
Yeah. So I think just a moment. Yeah. So when you're writing these APIs, you mentioned also that there's a way that you co-opt unsafe to enforce other invariants, essentially documenting things that the borrow checker or just Rust's type system can't guarantee for you, but you need to in, you need to uphold as a consumer of an API in order to maintain memory safety. Could you describe a bit more about situations where that's come up?

Participant
Yeah. So I'm going to pick my most prototypical example of this. So are you familiar with the exact size iterator versus trusted length iterator?

Interviewer
No.

Participant
Okay. Uh, let me send you some links real quick. So there's this trait in the standard library called exact size iterator. And it's basically a mistake. It purports to be an iterator that has a length that is known exactly. And it sounds like you can ask it for its length and it'll give you its length, but that's not really what it does. It gives you a length and that length may not actually be the proper length of the iterator. And people only realized this like after the fact, like, oh, we didn't put in a requirement that the length returned from this is actually the length of the iterator. And so there's this new trait called trusted length that actually does that. And it's an unsafe trait. So we end up with these two APIs, these two like iterator traits that we would use to control or like understand the length of the iterator introspect on it. And so you can end up in this very interesting situation where you can ask someone for a trusted length iterator and then whether or not you exhibit undefined behavior when they give an incorrect answer is kind of up to you. Like you could treat someone's, uh, I'm trying to think of a good example. Let's say you ask for, uh, someone gives you a trusted length iterator. They've either implemented it correctly and it gives you the right length or they've implemented it wrong, gives you the wrong length. Uh, you might only exhibit undefined behavior if you're doing unsafe stuff yourself. Uh, if you just iterate through the iterator, if you like with capacity of back to precise to some amount and you just push all the elements back, you're not going to run into undefined behavior even if they implement it wrong because you're still going to push them all back. You're using it basically as a hint. Uh, but if you do like a bunch of like raw allocations and you like offset pointers to write stuff, then there's a really high likelihood that you could run into undefined behavior if they implement it wrong. Uh, and so there's this weird notion where we have the length of an iterator being this thing that's unsafe to implement, but it doesn't directly correspond to memory safety. It's like you can construct situations where it can affect memory safety and therefore it's unsafe. Uh, it's kind of that weird dichotomy where, uh, you can bound things really tightly with unsafe and like build these really tightly knit systems of invariance. Uh, and you, you can, as long as you can like feasibly construct a situation where violating them would cause memory and safety, you can just mark them all unsafe, even if that never actually occurs in practice.

3:19 Uh, and so there's this...
As Documentation
Contract or Invariant

Interviewer
Gotcha. So you're just to clarify, this is something where if you interact with the API in a way that is not using unsafe, you are guaranteed to not invoke undefined behavior. But because there's this question of trust about the validity of the length, you still mark the trade as unsafe because you want to sort of be really conservative in case someone is using that in an unsafe context.

Participant

Yeah. Uh, that's, that's exactly right. Uh, we have this notion where unsafe means there are invariants that you need to uphold when implementing this trait, but that idea is really distant from memory safety explicitly. It's kind of like there is this, uh, logical bridge you have to construct between the two where violating these invariants can cause memory unsafety. Whereas when you're writing an API, you mark things as unsafe as soon as they require special invariants, but on the implementing end, you're, you like, may or may not leverage those invariants or you may or may not have a situation where violating them causes undefined behavior.

3:20 Uh, we have this notion...
As Documentation
Exposing an U...e API t

Interviewer
Gotcha. So this next question is a bit more specific to certain types. So I notice, um, there's just in your response to the screening survey, you mentioned using Box, Rc/Arc, UnsafeCell, and Cell in unsafe contexts as well as, um, MaybeUninit and ManuallyDrop. So could you describe how these memory containers are useful to you in certain unsafe situations and like how you reason about memory safety with them? In particular, some of the situations I'm thinking of with this question are related to Rc/Arc and Box. Like if you take a Box and unwrap it as a raw pointer, then you need to have the invariant that you eventually rewrap that in the Box so it can be dropped. Um, are there other situations like that with these containers or similar problems that you've run into in terms of using them with unsafe code?

Participant
Uh, I'll give different examples for Box and Rc/Arc. Uh, so I do windows programming and windows programming is absolutely wild. Uh, when you spawn a window using windows, uh, like create window API, uh, you can give it basically a data pointer to pass along and associate with that window. And so the natural thing in Rust to do is just Box something into raw it and stick that pointer into that API, like pointer sized slot that you have, and then you can pull that back out to reference your data. And then when the windows closed, you get an event and you rewrap it and drop the Box to be polite. Uh, so that's a case where like I use unsafe stuff with Box. With Rc/Arc, I've written a, uh, an executor, uh, and, uh, it uses tasks that may be shared in multiple locations, like wakers may reference a task and like your execution queue and your executor may reference a task. Uh, the thread that's actually pulling it may reference a task. And so the natural way to express those is with an Rc/Arc. Uh, but there is a tricky bit where, uh, I can't remember exactly what it is. I think it's, uh, you only have like a pointer sized, uh, slot to hold memory in. I think it's, uh, it might be waker, uh, lets you create a waker from and a pointer to unit, uh, using an unsafe API. And so naturally I just make a task, put it in an Rc/Arc, and then, uh, turn that Rc/Arc into a raw pointer to the Rc/Arc, cast it to that unit type and throw that into the waker API. Uh, so that's a place where I've used Rc/Arc for all of its like unsafe stuff. And then it's easier to just like manually increment and decrement the ref count and stuff.

3:2...
Windows
3:22 And so...
Allocation in Rust
Box<T>
3:24 With Rc/Arc, I've written a,...
Arc<T>/Rc<T>
3:23...
Easier or More Ergono

77 **Interviewer**
Gotcha, okay. And then with MaybeUninit and ManuallyDrop, could you describe some of the situations that you've used those?

78

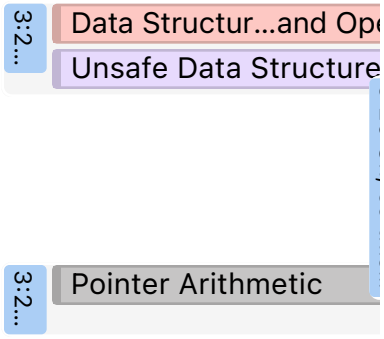
79 **Participant**
Yeah. Uh, let me send you another crate. I'm sorry, I write so many crates.

80

81 **Interviewer**
You're good. This is great.

82

83 **Participant**
So I write this other crate called [name], uh, which lets you do safe destructuring of a whole bunch of different types. And among those are MaybeUninit, Cells, and ManuallyDrop. Uh, so this is essentially for doing, uh, field by field initialization for some structs. Uh, you would basically make a MaybeUninit of it and then you want to like destructure it into MaybeUninits of all of its fields that you can initialize those in place. And then once those are all initialized, you just assume in it the whole thing. Uh, so, uh, yeah, it's pretty straightforward. It's just turning my MaybeUninit into a pointer, doing offsets, a whole bunch of stuff like that.

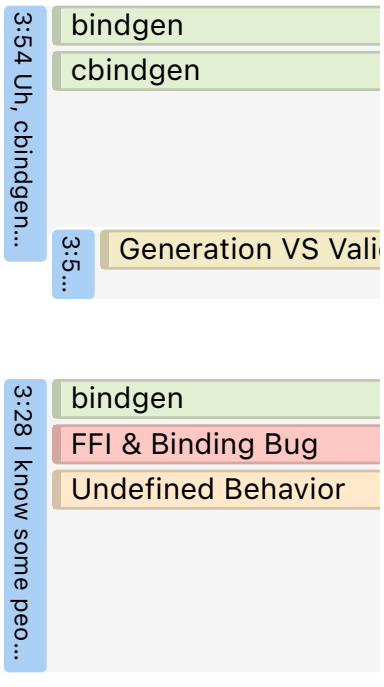


84

85 **Interviewer**
Gotcha. Okay. So then related to use of foreign functions, you mentioned doing bi-directional FFI usage. So with C, C++, JavaScript and TypeScript. Um, and then you use bindgen, cbindgen, wasm-bindgen and windows-bindgen. So in particular with, and as well as just writing bindings manually, could you describe about your experience in writing foreign bindings and using those tools for each language?

86

87 **Participant**
Sure. Uh, they're all very dramatically different from each other. Uh, cbindgen and bindgen are, I mean, they basically generate what you'd expect. It's like, what is the signature of this according to C and C++? And then, well, according to C, I should be, I should be more careful with my wording. Uh, and that basically just gets exported. It's almost no different from just a raw header file that lists out all of your symbols. It's basically for convenience. It's not so difficult to write all those by hand, but keeping them in sync is a pain. So that's, that's how I would describe my experience with them. They're, they're pretty straightforward. And you can like configure them to do a lot of extra stuff, but I generally don't end up doing that. I know some people do, but one of the hazards of doing a bunch of like custom stuff with bindgen is if you like add derives to your, or like attributes to your bindgen configuration, you can cause all sorts of bad things to happen if you like tweak the layout properties of the type in a way that's not, uh, consistent with like, uh, repr(C) or something. Or if you're going to derive some stuff for that type and it ends up that, oh, well, you didn't consider this one case where the type has some awkward layout and now the derive makes code that is, uh, like either faulty or like,



we'll, we'll result in a compiler error or we'll cause undefined behavior when you do stuff. That's a lot more rare, but I've seen some cases where it happens mostly with like custom derives that aren't meant to be like bulletproof. Um, for wasm-bindgen, uh, I basically have only used it for tests. I just mark everything as wasm-bindgen and I fire it off to whatever the wasm thing that runs my stuff is, and it eventually gets back to me or gives me some cryptic error that I have to go search on GitHub for. Uh, windows-bindgen is, it's just generating, uh, bindings for Windows APIs. Those are a lot closer to like C bindings for the Windows APIs, but they're generated from metadata. So you pick and choose what APIs you want.

3:2... bindgen
FFI & Binding Bug
Undefined Behavior
3:55 Uh,... windows-bindgen

Interviewer
Oh, so yeah, very different experience that you're describing with each one of these as well as different use cases.

Participant
Yeah. Oh, I didn't mention, uh, the JavaScript. Uh, I have used wasm-bindgen to actually communicate back and forth with like a JavaScript application. That was extremely painful. Uh, I did not expect it to be so difficult to like just make calls back and forth between Rust and JavaScript, but it required a ton of boilerplate. And even then it was like, man, this is a lot more complicated than I wish it was. Wish I could just make like one call. Uh, I think part of that might be that I didn't use some of the like more sophisticated ends of the, uh, like JavaScript generating parts of wasm-bindgen. I think there's like something it can do. It can like generate you a JavaScript file that does the bindings on the other side. Uh, but if there was, I didn't figure out how to do it. And I was just like, I just want to call one function and then like print it to the console so I can make sure it works. Uh, that's basically what I use that for. Um, I'll give you, I'm sorry I'm sending so many links. Hopefully you enjoy this.

3:30 Uh, I have us... Limitation of Binding T
wasm-bindgen

87 **Interviewer**
Excellent. I really, really enjoy links. Please send all the links.

94

95 **Participant**
Okay. So I wrote a blog post a while ago. Uh, I'll send the blog post and I'll send like the meaty stuff. This was the blog post, but for it, I made this, uh, JavaScript puzzle game. And part of that is it has a solver for the puzzles that's written in rust compiled to, uh, web assembly and then loaded in and invoked from the puzzle. So it like takes the state of the puzzle, which is all, that's all JavaScript stuff, turns it into a format that the, uh, the Rust side can understand pumps it into the, uh, the solver in the back end and then it returns basically like a solution for it. Um, oh, I didn't send you the, uh, you have to add a little thing to the end. Uh, gosh, I don't remember what it's called. At the end of the article, I give you a link to how to turn on the solver. Oh yeah. The very last puzzle in that article, if you scroll all the way down, uh, there's one with the solver in that. And then, uh, oh yeah, if you add and controls equals true, that'll turn on the solver for, like that, that should do it. Yeah. Question controls. Didn't realize I could edit stuff in. Yeah. So that was really fun. That was like,

3:31 This... Web Application Devel
... Rust Performs Well

that hit the nail right on the head of what I wanted WASM to do for me, uh, because writing this kind of a solver in JavaScript is completely untenable. The like memory performance is not high enough in a GC language.

3:32 Tha... Rust Performs Well

Interviewer
Gotcha. So related to the memory models of these languages that you're using, have you had any challenges when you are working with certain assumptions and rules about rust and then passing that memory into a language that has a different model and different assumptions?

Participant
Uh, so basically no, but that's only because the guarantees are so uh, ill-defined that I basically only stick to like extremely primitive stuff back and forth. Um, so if you're familiar with CXX, uh, CXX creates C++ bindings for Rust, bi-directional, so you can call C++ and Rust code like from each side. Uh, and there are a slew of like really tricky correctness problems because Rust's memory model is not precisely defined, but also is certainly different enough from C and C++ that you can't just throw objects over the fence. Uh, so I'll say in my personal experience, I haven't run into issues with it, but only because I basically avoid that, but I'm aware of a lot of really tricky issues with, uh, C++ and Rust interop. Uh, C generally tends to be less of a problem because the memory model is so simple, but, uh, for example, pointer provenance is a really good example of where C and Rust diverge because Rust doesn't exactly have strict provenance, but it's moving in that direction and strict provenance is a lot stricter than C's, uh, forget what that acronym is, but I think it's PVNI, something like that, uh, provenance via non-integer, whatever, uh, and so if you call any C code, uh, you basically have to, uh, understand that you're working inside of a different memory model where like, uh, pointer and pointer round trips are safe, uh, and that's not necessarily the case in Rust. So, uh, yeah, this isn't resolved yet, but it may be that eventually you'll have to turn on a compiler flag to use C's, uh, provenance model if you want to do foreign function calls to C.

3:56... Simple FFI

3:33 and there... Different FFI Memory
Shifting Ground

3:5... Shifting Ground

95 **Interviewer**
Okay. And so that that provenance issue seems to be the overarching problem you're describing at C and C++. You also mentioned when you did the WASM interoperation that the solving you were doing for the puzzles was just not feasible in a garbage-collected language, aside from performance, were there any other GC-related issues or just JavaScript-related issues that you had when interoperating between WASM, JS and, uh, and Rust?

102

103 **Participant**
Hmm. Uh, I think most of it was just getting all of my, like, uh, on the JavaScript side, setting everything up to make a function call into WASM is kind of tricky. You basically load everything into, like, uh, JavaScript's version of, uh, like, raw arrays and then pump them into, like, a function that you dynamically fetched out of your, like, loaded Wasm binary. Uh,

3:35 on the... Limitation of Binding T
wasm-bindgen

and so it's really weird. Uh, it's kind of annoying to write all of those, but, uh, yeah, like I mentioned before, it's probably something that, like, uh, Wasm-bind-gen could do for me but I didn't figure it out in time, uh, or I didn't care to figure it out. So that part in particular was a little tricky.

3:35 on t...
Limitation of Binding T
wasm-bindgen

Interviewer
Gotcha. So now we'll get back to a higher level. And in, for this question, I want you to consider sort of any of the unsafe usages that you've talked about before. So for performance, for maintaining invariants, for the FFI stuff. Um, so describe a bug that you found that involved use of unsafe and talk about how you discovered it, whether that was hard or, or easy as, as well as how you solved it.

Participant
Okay. Uh, how tricky do you want the bug to be?

Interviewer
As tricky as you've got.

Participant
Okay. Uh, I'll give you, I'll pick two of them. Okay. So here's one for the standard library that I identified. Uh, and this was basically that there are currently two in the future, there will be like, I think three different APIs for performing allocations. Uh, and the safety requirements for them don't 100% align, uh, even though they are structured. So that calls to one eventually just boiled down to calls to the other one. Uh, and so this is all like described in detail in the bug, but the safety requirements for allocator grow, don't check, don't, don't require that the allocation size meet some certain bounds. Uh, even though the thing that they call do have that restriction, like the inter call requires that it be within a certain size, it doesn't check those to like panic safely. And so, uh, you can construct like a custom allocator that meets all the requirements and then plug that in and then make an allocation that also like on the other end meets all the requirements that they're relaxed because you're using the like looser API and it causes a segfault because you've tried to do the wrong thing and blown everything up and you've got, you've gone and violated my, uh, safety guarantees without realizing it. Uh, so that's the issue. I found it by basically poring over the allocator code because I was writing my own version of it to be like stable that I could ship with a library. Uh, and I realized just by chance that the allocation APIs didn't have the same safety requirements. And so I reported that upstream. Uh, I constructed an example where that could cause undefined behavior and then reported that upstream. Uh, so that's an example of it in the standard lib. Uh, I have tons of them in archive because every time I turn around, well, not every time I turn around, but there have been a lot. Um, trying to find one that's not a panic though. Okay. Uh, I do remember some stuff with like, I wrote a custom vector class and it could have exhibit undefined behavior. If you turned it into a, yeah, here's a good one. I have a specialized class for highly aligned byte vectors. Uh, so it would make sure that your bytes are always aligned to alignment 16. Uh, that's for

3:36 Uh, even though the...
Logical Error
Requirements Bug
Undefined Behavior

3:58 Uh,...
Engaging with...st Com

some crazy stuff down the line. Uh, but you had, I made an API surface to turn your highly aligned byte vector into just a regular back of U8 and the original implementation just, uh, into raw, uh, my original one and then from raw, my new, uh, slowly aligned allocator. And at the time, which has completely slipped my mind that when I free that allocator or when I drop that, that new Vec, uh, it's going to free using a layout that is the right size, but it's going to say alignment one because that's the alignment for U8 when I allocated it with the rights with the same size, but alignment 16. Uh, and so someone found that and reported that. I think they found it with Miri, uh, and reported that. And, uh, it turned out to be a very sad, but easy fix where I just allocated a new allocation and copied all the bytes over.

3:37 And at the time, whi...
Allocation & Alignmen
Improper Alignment
Miri found a bug

Interviewer
So you mentioned that you've had a lot of examples of these types of bugs that you found. Are there any patterns that you've seen throughout all of them?

Participant
Hmm. I would say that kind of, yes, I would say that it's really diverse, the kinds of memory safety errors that you can get, but there are like clumps. A really big clump is obviously like just good old fashioned. You mismanaged your pointers, uh, uh, you know, using the wrong pointer somewhere or using it after you should or leaving a dangling or, you know, all sorts of weird pointer errors. That's obviously a big clump of them. Uh, another kind of interesting clump, uh, is, uh, violating Rust's kind of peculiar semantics for certain primitives, like allocation, I mentioned. Uh, you can't allocate or you're not supposed to allocate. Uh, you're not supposed to make an allocation of size zero in Rust that is undefined behavior as defined by the API. Uh, let me make sure, let me make sure that's right. I think it's allowed to return a null pointer, uh, allocator allocate. Uh, let's see. It might just be that allocating with size zero is not defined behavior, but not undefined behavior. Alloc. See global alloc. Okay. Yes. Okay. It's not the allocator trait. It's the alloc function, which calls out to global alloc, alloc, and that safety condition says, uh, you have to make sure that the layout that you pass has non zero size. Uh, and that can trip people up if they're not ready for it. Uh, if you've written a generic container and your type is a ZST, then, uh, you go to like allocate some array of them, you'll get a, uh, length zero, uh, layout in there, and that'll trip up your allocator potentially, potentially it will trip up your allocator. I've seen plenty of cases where it doesn't on certain systems and things just chug along as normal. Uh, but, uh, I've also run into situations where it introduces undefined behavior for real and it just, you don't realize that it happened until it's way too late and then tracking it back as a nightmare. All right. So that specific one has happened to me where I've allocated a, where I've made an allocation of size zero and it caused undefined behavior and I had to like basically bisect it back because I couldn't corner it. It wasn't like causing a segfault immediately. I had to bisect it back until I had to figure out what was causing it. So those are two big clumps. Uh, there is this kind of weird badlands of undefined behavior that is related to like semantics, uh, in Rust that are not exactly

111

3:59 A really...
Pointer errors

3:38 It's the alloc function, which...
Allocation & Alignmen
Architecture D...dent P
Zero-Sized Allocation

3:60 I've se...
It's not a problem

3:61 So that...
Allocation & Ali
Zero-Sized Allo

3:3...
Shifting Ground

defined yet. Uh, stuff like uh, pointer provenance is a really good example of this, but uh, there's also like the whole memory model is not well defined. And so, uh, there are things that are okay by convention, but might technically allow for undefined behavior. Um, and so like the unsafe code guidelines have like slowly been like tweaking this way and that way to make sure that they like tighten in on what is okay or not okay in Rust. Uh, if you make a mutable reference from a, uh, an uninitialized value, like if I have a MaybeUninit and I have a pointer to it and dereference it into a reference, uh, is that undefined behavior? The gut reaction is yes, but there's a lot of question about, hey, what if I want to pass in, uh, for example, in [crate name], you have this byte structure that's used for like an input output buffer. What if I want to make a bytes that, uh, aliases mutably those uninitialized bytes? Uh, but I'm only going to write to them. So it's basically not going to cause any problems when it's boiled down to like LLVM IR. Uh, but still it's not exactly in alignment with what we say things should be undefined behavior and having me recheck that all of your objects are like recursively initialized is also really tough. So there's this ongoing discussion of like, should we allow mutable references to be uninitialized as long as you don't read from them first? And, uh, it's just a question, you know, no one knows the answer. Right now it tends to be the case that it's not going to cause undefined behavior unless you read from it. Uh, and so that's kind of where everyone is going with it. Uh, yeah. So lots of weird stuff going on over there.

3:39 Uh, there i... Shifting Ground

3:41 Uh, if you mak... MaybeUninit<T> Shifting Ground

3:40 So it's basically not... Shifting Ground

Interviewer
Interesting. Yeah. So then related to bug finding tools, you mentioned that you've used Clippy and Miri. Could you describe your experience with those tools?

Participant
Oh yeah. Uh, Clippy is great. Asterisk. It's definitely annoying, hence the name. But there have been a couple of times when there's legitimately caught bugs for me, and that makes it all worth it. Uh, I can't think of any particular ones, but I think it's all related to like, oh, you used the, uh, the wrong semantics here. This is going to cause an implicit dereference and you're not allowed to do that right here. You probably didn't mean to implicitly dereference this. Uh, with Miri, Miri is just like the big hammer for finding undefined behavior. Uh, I run all of our tests through Miri and if stuff comes up, it's undefined behavior. And then whether or not it actually is undefined behavior, we fix it. One of the tricky things about Miri is that stacked borrows is a like runtime approximation of Rust's, uh, ownership rules. It's not exactly correct because Rust's, uh, implementation is like too rapidly changing at like the micro level to like formally specify what it is. Uh, but stacked borrows is like a really, really good approximation. And so for most people, if stacked borrows throws up an error, you have a problem. Uh, for me, Miri got on my nerves because, uh, stacked borrows didn't handle relative pointers where you create intervening references properly. Uh, it basically always detects that as undefined behavior. Uh, and that really made me sad because I wanted to use Mary to check everything else, but it would just instantly error out as soon as I, uh, did like one thing with a relative pointer. Um, the nice thing is we have a new

3:42 Oh yeah. Uh,... Clippy Finds Bugs

3:44 One of... Shifting Ground

3:45 Uh, for me... Stacked Borrows is too

model that's like currently in development called tree borrows and that handles the case perfectly. And now I can run everything through tree borrows and it all passes. It makes me very pleased. Gotcha.

3:... Stacked Borrows is too

Interviewer
Gotcha. Yeah. Interesting. I was also following tree borrows. So that's, that's interesting to hear that that solved your particular problem.

Participant
Yeah. Uh, I was talking to, uh, [name] at [event] last year and I was basically like, please, can I have some sort of fix for Miri? And he was like, I don't know, maybe. So I'm glad that it happened.

3:63 I... Engaging with...st Com

Interviewer
Gotcha. And then for the final question, with these tools, do you feel like they have handled all of the problems that you faced right again, say for us, or are there existing problems that your development tools are not adequate for?

Participant
Do you tell me more what you mean by like problems?

Interviewer
I would say bug patterns or undefined behavior, crashes, cases where you've seen undesirable behavior in your program, but this hasn't been something you've been able to catch easily with something like Miri or Clippy or a sanitizer.

119

131 **Participant**
Got it. Uh, you cannot fix undefined behavior with just Miri and Clippy. They are great tools, but they are not enough. Uh, from personal experience, I can say you need a good debugger, and it is the most important component, because when push comes to shove, only a debugger will tell you what's going wrong. Uh, so I had a extremely nasty problem with yet another crate that I've written called [name] that does like in place validation of arbitrary bite slices. You can also link that. Uh,

3:46 Uh, yo... Debugger

132

133 **Interviewer**
Gotcha, yeah, that'd be great.

134

135 **Participant**
Uh, this one. So because of the very nature of what it's doing, there is a ton of unsafe in this. We basically have to like probe some data on disk and make sure that all of the invariants that we need are upheld, whether that's Chars being in the right range, Booleans being zero or one, floats being some restricted set. Uh, and then eventually that builds up to like, Oh, well, you know, vectors need to have valid pointers and this and that.

3:47 So becaus... Contract or Invariant
No Other Choice

And hash maps need to have, you know, entries that hash to the right values back and forth, into the right buckets. So it gets to a really high level really quickly. And sorry, I lost my train of thought. I was originally talking about debugging. Okay. Yes. There's a specific issue on this that is solved now, but this one, this particular issue was related to, uh, this library that we were using for UTF-8 validation called SIMD UTF-8. And through no fault of their own, they introduced this really nasty undefined behavior, because of a mislink on Windows when you use thin LTO and figuring out what this problem was was really tough. It basically ended up being like an atomic pointer somewhere was initialized to the wrong value. And it was because the like section that it was being linked in with wasn't being relocated properly. Really crazy. And the only way that I could figure it out was by firing up a WinDBG, which is Windows' reversible debugger environment. And going all the way up to the point of the crash, examining the memory location, rewinding to the beginning, finding that it was exactly the same value as it was, and there was nothing changed in between. And then saying, well, it's initialized to the wrong value. That's the problem and reporting that upstream. So like when push comes to shove, you need a good debugger. I will say I don't think that we have really a rust quality debugger experience yet. And I think everyone basically agrees. We have a lot of debuggers at work, like MSVC's debuggers works and LLDB works, but it's not really like a rust level experience where you can get all the bells and whistles that you want and things just work nicely together. It's still kind of just getting down in the mud.

3:47 S... Contract or Invariant
No Other Choice

3:48 And through no faul... Architecture D...dent P
FFI & Binding Bug
Incorrect Linking
Undefined Behavior

3:49 So like when pus... Debugger

Interviewer
Gotcha, interesting. Yeah, I definitely know what you mean by a Rust quality experience there, yeah. Yeah.