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

Participant
So I was motivated to learn Rust in 2012, which was a very, very long time ago and honestly, so I was a TA for an OS course and I saw so many students make so many mistakes in the kernel. I think at some point that I was like, oh my God, C is like just kind of scary. So basically, I was motivated to learn Rust. I was just interested in language at the time, but interested in a language that solved problems that I was running into like member safety, but also having a nice performance in the level of NSFC. So as my initial learnings wasn't exactly the language at the time, but it has since evolved into language as of now in 2014. So very much basically my motivations are low level performance, low level control, high degree of safety, really hard to mess up.

13:1 So b... Memory Safety
Rust Performs Well

Interviewer
Gotcha. When are you generally like applying Rust for that requires those things?

Participant
My job is for, gotcha. So I, so three, personally, I was, I was working on Rust for a very long time, but nowadays I work on [tool], which is a WebAssembly engine for written in Rust. And so we as a kind of language runtime do a lot of really low level things because we want [tool] to run as fast as possible. We're doing funky things like we're compiling code and then actually executing it. So it's like, I don't know, doing pointers and stackwalking and all that fun stuff. So for us, a very low level of degree of control is very important, but at the same time, we are running untrusted code. So a very high degree of safety is also very, very important to us. So we have, one of the primary reasons we're using Rust, for example, is that the API that we give to users for the, for the [tool] is 100% safe, I mean, I guess there's escape hatches, but by default, 100% safe, and that's very, very valuable where even not only internally can we use safe code, but externally everyone using us can also rely on all the safe code.

13:... Compilation & Interpre
13:... Compilation & Interpre
13:4 S... Memory Safety
13:6 a... Preference for Safety

Interviewer
And then what, I guess, in this case, are you using unsafe for?

Participant
Quite a few things internally. So I've done a lot of unsafe for us historically, but at least currently with [tool], a major one is that we are generating machine code and then executing it. So unsafe is obviously required there just because you're asserting the output of the compiler is correct, but we're also doing quite a lot on the runtime side of things in terms of just kind of pointer tricks. Like we've got a lot of aliasing, a lot of things that are not really modeled very well and strict safe, like we can't

13:7 Quite a... No Other Choice
13:8 b... Arc<T>/Rc<T>
Box<T>
Increase Performa

just use Rc, we can't just use Box, things like that. So that's like the runtime speed kind of low levelness. And then the other aspect is also very much, I saw in the question, there's a lot of questions about bindings, which is that we do provide a lot of bindings to other languages. So we provide a CAPI, we then bind that in Python and meta languages, and then, I guess, those are sort of like the main things here and there.

13:... Arc<T>/Rc<T>
Box<T>
Increase Performance
Multiple / cyclic aliasin

Interviewer
Gotcha. So just to summarize, three main applications. First you're executing untrusted machine code, which you say would just like inherently require unsafe to do your, oh, sorry, you go,

Participant
Not so ...

Interviewer
Sorry

Participant
...not so much untrusted machine code, a trusted machine code, but from an output of a runtime component. So like the wasm is untrusted, but the output of the compiler is trusted. So like we can, we can't trust, we like basically WebAssembly is like guaranteed sandbox semantics to start at a language level, but then we're running it through a compiler which claims that it actually compiles a wasm successfully, but it can't have bugs. So effectively, like we've had bugs in historically, we have the reason to believe it's incorrect, but there are just bugs occasionally. So trusted machine code, but even still, I mean, that's just unsafe operation is just cast bytes to a function pointer and call it.

13:... Transmute

Interviewer
Gotcha. Gotcha. And then the second aspect is that there are certain usages of pointers and aliasing patterns that you can't express using some of the default memory containers. Could you talk a bit more about those?

Participant
Yeah, definitely. So for us, we have, like we've chosen us a couple of layers of modeling, but like the main thing for us is that we want to expose the ability where you can instantiate wasm instances within us, what we call store, and then they all kind of refer to each other. So you can take like the exports of this instance, hook them up to this instance, kind of push them all back and forth. And then once it's all executing in wasm, the actual machine code generated by our code generator called [generator] is interacting with all those pointers as well. So basically, we have tons of pointers pointing to everything else all over the place. So there is no sense of like, there is a unique owner sitting in one particular location. It's rather, well, each of your imports came from this instance, which has this pointer, but each of those imports came from that instance,

11

13:10 So bas... Multiple / cyclic aliasin

which has those pointers. So kind of lots of like internal pointers here and there. We still maintain like a very high level embedding layer. We can say like there's a unique point of ownership, nothing there, but internally kind of doing all that, the representations just to have a very low level representation. But the other main thing there is not so much like, it's can't be guaranteed by Rust per say, but like one example is we effectively have our own custom arc, which Rc/Arc in the standard library works perfectly fine. There's no reason to not use it. But for us, we have JIT code, which is generated by [generator], which is actually interacting with it. So the JIT code, for example, is incrementing and decrementing the reference count. And we can't rely on the in memory layout of the Rc/Arc in the standard library for very good reasons. We kind of effectively have our own version of arc, which is a slightly different in a few words, but primarily we know the in memory representation. So we know where reference count is and we know how to write JIT code to modify it.

13:... Multiple / cyclic aliasin

13:12 but like one example is... Arc<T>/Rc<T>

Unsafe Data Structure

Interviewer
Gotcha. so is it just that the standard library's Rc/Arc does not have a constant defined memory Layout?

Participant
So you have exactly if it defined, for example, that like the first the pointer pointed to an atomic reference count, which was this wide, and that was it, then we could do that. But then there's also stuff like with weak pointers in the standard library. So that's just like, basically, because the standard library doesn't do that, we ended up rolling our own check.

13:13 So you... No Other Choice

Interviewer
Are there any other cases like that where some restriction or feature maybe of the standard library gets in the way of what you're trying to accomplish, so you need to get around it using unsafe?

Participant
It's interesting because we, as of like just this week, we started running some of our tests in Miri, which is I'd like to try and verify some of the unsafe code. And that was actually very much motivated by last week, we had a CVE, just we had undefined behavior, one of our functions, which actually surfaced in real user behavior. So anyway, long story short, we were using Box of T for some data structures. We would say like, basically, we would put in a Box, which we would assume would heap allocated, we would then put the Box in the store, but then we retained pointers to it all over the place. So we wouldn't actually use the Box pointer. We just kind of like keep that rooted in a sense. But according to Miri, whenever you move a Box, you're invalidating all pointers that you derived from the Box. And so that was, it's not like a perfect example, like the Arc/Rc example, but it's one where we did want the property where we can move the pointer owner without invalidating previous ones. So we effectively made our own little Box lookalike style thing. That's a little less like egregious than the Arc/Rc, but that's another example of sort of where the standard library isn't quite suited. But I know that the, the

23

13:15... Running tests through

13:14 . But a... Borrowing or Provena

Stacked/Tree...rrows V

... Shifting Ground

unsafe code rules are a little influx there at that point. So it's one where Box might be providing too many guarantees that we don't actually want to give, or those are also under debate. So I don't know whether that's the correct thing. I won't pass judgment on that. That's just an example of what we're going to.

13:17 But...
Shifting Ground

Interviewer
Gotcha. Really interesting. Um, yeah. I, is there a, so Box and Arc, are there any other situations you can think of? Or are those the primary two that you think are representative of the pointer tricks that you'd be doing in context?

Participant
I think the other thing is like, probably the best way to explain this is like, imagine a graph data structure where on the outside you have just an owned API. It's all perfectly safe for us, but internally everything's just got pointers to everything else, kind of like a double link list of a sense of like, you just got pointers to your next to your pre they've got pointers to you. Someone else has pointers to you. That's sort of like kind of what our internal data structures look like. So it's not so much that the standard library is really in conflict there. It's more of just like, there really isn't a standard library data structure for that, which makes sense. There's not really a like statically safe paradigm here. It's more of like, we just kind of got to guarantee everything internally.

13:18 imagine a gr...
Extensive, Non-local U
Multiple / cyclic aliasin
Unsafe Data Structure

13:19...
Runtime Assertion

Interviewer
Gotcha. Would you say that these structures generally take the form you described like a double linked list or like, I guess, could you describe a in more concrete terms, what one of these data structures would be laid out as and what the pattern is that it uses in particular that goes beyond the safe rust borrowing semantics.

31 **Participant**
Yeah, definitely. So this is like, so it would be simply, if I talk about web assembly, are you familiar with us and we're should I give it like a brief intro on that aspect?

40

41 **Interviewer**
I'm familiar.

42

43 **Participant**
Yeah. Okay, cool. So for each web assembly instance, we have an allocated set of state and that state includes like the function state. It's got some globals. It's got some other various stuff. But like, I'll talk about functions are a big, big example. So one of the data types in wasm right now is a FuncRef, which is just a pointer to a function. And we represent that as a literal pointer to a function or it's actually a pointer to some data and memory, which one of those elements is an actual literal function pointer from [generator]. And so one of the contextual pieces of a

FuncRef is it doesn't just close over code, but also the instance data that comes from. So if you capture a FuncRef, then your function, then like the function you're capturing has access to its globals, its memory, its tables, all that stuff. And so basically our instance state, which we call VM context, internally as represents one instance, and then every FuncRef for an instance is effectively [generator] plus a VM context. So it's kind of like a function pointer and the data for the function pointer, both of which are pointers. So for us, a FuncRef is a pointer to an in-memory data structure, which has two pointers, which is the function and the code. So that's like the basic data structure for this. So this means that my instance VM context, which has pointers, which has FuncRef inside of it, are all self-pointers. So all of those are pointing to the VM context itself. So you got self-pointers going back and forth. And then additionally, whenever I take my FuncRef and give them to a different VM context or like a different instance, they're going to point back to me. And so they've got pointers over to me. I've got pointers to myself. I've got a pointer to myself. And then additionally, I can also have pointers back to you if you hand FuncRef to me, because everything can be cyclic once you actually get into like the runtime semantics of all this. So we end up having this soup of just every instance can have pointers to other instances. And there's not really any limit to it. And they're all kind of self-pointers as well, where every, like every VM context has a lot of self-pointers to its own self VM context and things like that.

13:46 So this means that my ins...
Extensive, Non-local U
Multiple / cyclic aliasin

Interviewer
Gotcha.

Participant
I don't know if that's concrete enough or yeah.

Interviewer
Oh, no, that's, yeah, that's perfect. That's perfect. So you mentioned earlier there being some level of separation between a safe interface and the space in which you're using a lot of those cyclic pointers. Could you describe a bit more about how this is encapsulated and in general, like how you're encapsulating unsafe and the code that you work on?

Participant
Definitely. So this is where we have the [tool] crate, which is the embedding take the official embedding API [tool]. And I see official in the sense of like, there's lots of little crates underneath that, but they're not like, we don't intend for you to use them. It's just kind of how we organize our code. So the official embedding API has no unsafe code in it or rather you can use it without unsafe code. So you can call wasm, you can instantiate wasm, you can compile wasm, nothing unsafe there at the top level. So that's been our guarantee the entire time is like, when you talk to it and when we give it to you, we are giving to you a hundred percent safe API. But then once you get into the internals, everything kind of all bets are off. We got lots and lots of unsafe code. And this is one where like, we don't actually do a great job of organizing this right now, which is that

43

13:47 So the offici...
Exposing a Safe API
13:...
Difficult to Encapsu
Unsafe is Diffi...to U

ideally there's like a very clear boundary of you can, I mean, the theory is every single time you write an unsafe block, you can have a nice little comment above it saying like, this is obviously correct and you as the reader can verify it because you can just kind of like check a couple of variants. But for us, I think we checked recently, we have like 500 unsafe blocks and like 1500 uses of the word unsafe. So it's a very much all ends up being kind of interrelated to everything else. And it's very difficult to reason about one unsafe block in isolation. So I guess two point of like organization is that the top level part is organized in the sense of no unsafe. We give you an API which we claim is safe. It's kind of tough to verify that. It's kind of the question of like, how do you prove that Vec is safe? I mean, like given it's safe API. So, but like, I mean, that's our theory that we've done that. But then internally, there's honestly not a ton of organization. There's a lot of really implicit guarantees that are just kind of carried around between, between everywhere and you, it's really unfortunate, but you do have to understand a huge amount of the system to mutate part of the system to note, to note that something messed up unfortunately.

13:20 And this is one...
Difficult to Encapsulat
Unsafe is Diffi...to Und

13:48 We give you an API...
"Safe" API
Unsafe is Diffi...to Und

Interviewer
Gotcha. There are two really interesting parts there I want to touch on. First in the way that you described the interface itself, to a certain degree, you mentioned that you sort of guarantee a safe interface, but that guarantee is not complete necessarily. To what extent do consumers of the safe interface need to meet certain preconditions beyond what they would meet just by passing in safe rust types?

Participant
So I should clarify there. And I should say, so I mentioned Vec as an example. So for example, Vec has a function called setlen, which all it does is it just sets a field to the integer you provide. Now, this is inherently unsafe because when you combine it with like iteration or indexing, if you, if I suddenly say the length is bigger than it actually is, that's like fundamentally unsafe. So what could be a safe API is actually unsafe. That's why the setlen function is unsafe, obviously. So along these lines, we have asserted that the current API was in time is 100% safe. So no preconditions necessary. No matter what you do, if you don't use unsafe, you literally cannot cycle. You get determined to six semantics. Like we can still infinite loop. We can still crash your program. Like actually, can we crash your program? I don't think we can. No, no, no, not crash. It's just infinite loop. Infinite loop is the worst one and resource exhaustion. Like we could probably allocate a lot of memory if you don't live it, things like that. But sorry, I don't mean to say that, like, we don't have any preconditions along the lines of like, if you use the safe API, you still have to make sure that you do these other things and things like that. We very much try to give you the guarantee that no unsafe, perfectly. We'll never crash. Only might consume too many resources.

Interviewer
Gotcha, gotcha. And then the, well, like one practice you hear from the

Rust community seems to be that you want unsafe to be very minimal and encapsulated. So that you can reason about it in isolation. But from what you describe, it seems like it's very difficult to be able to modify or change or sort of audit a particular piece of unsafe in that the structure of this sort of internal interior and safety is very like in mashed together. Is there a, do you feel that that architecture is just utterly necessary as a fact for the type of application you're working on? Or would there be a pattern that you could take to avoid that and to make the unsafe more easy to reason about in isolation, I guess?

Participant
I would not claim it's necessary, but I will say I don't have a better way to organize it. So it's one of those where like, like the communities, like just general idea here of like, encapsulated on safety, perfect. That's exactly what we want to have. We very much agree with that, but we just have not found a great way to manage that. And there are some prevailing things like, I'm probably exaggerating a little bit when I say that you have to understand like everything to change anything. But there's, there's, there are like some kind of prevailing architectural design decisions, which if you don't understand them, most of them, most of [tool] would not make sense. So like, I mentioned the store earlier, where like a store is an own thing. If you don't understand that like everything, like once you put something in a store, it just lives for the entire lifetime of the store and it had never been deleted. That's a very critical lifetime invariant, which we have across every single component, but it's very easy to miss that. Like that's, like that assumption is sort of enmeshed and just the structure of APIs, the structure of things. And it's very difficult to have like the one location, which is like, okay, by producing this, we are like giving you a safe type, or the safe type can only be dereferenced in the context of the store. Like we could in theory do something like that, but then it ends up a little adding extra runtime overhead. And so once we're in the internals, we don't want to add runtime overhead. So I guess to answer your question, we have not figured out a way without runtime overhead to encapsulate the unsafety in the last time, even more than it already is.

13:49 So... Preference for Safety

13:21 But... Audience-specific Rea

13:22 So like... Contract or Invaria

13:50 And it's very dif... Increase Performance

Interviewer
Gotcha. Could you just follow back and get a bit more clarification? Could you explain the source of the runtime overhead again? And like, like what in particular that would mean for your implementation if you were to change it?

Participant
It's very hypothetical. I'm not really talking about anything concrete here. It's more of like, you kind of like one of the examples of the unsafety is that a lot of data is stored inside of a store and it lives for the lifetime of the store. So a very naive answer to that would be like, okay, we'll put the data in the store, give me an index, and then whenever I want to access the data, I present it to the store. It checks that it's the right index and it gives me the data. Like that would be a very common way to say like, okay, that's perfectly safe. Don't have to worry about that. But if you did

that, then you would have a lot of, then we would hypothetically have runtime overhead in terms of a lot of index checkings. Like every time we have a pointer, we have to pass it back to the store or things like that. I should say it's not purely runtime overhead. We are pretty constrained in the sense that we are generating dynamically code from like [generator] to interact with the actual runtime data structures we have as well. So the interactions are not only what we wrote in Rust, but also we are generating code to interact with. So for example, these VM context pointers that are kind of sitting all over the place, if that was actually an index within the store, then the layout of the store would have to be known to the JIT code. So it would say like the JIT code itself, when it access the store would do the bounds check would then index and do everything manually. So it's like, we couldn't write necessarily safe Rust code for that. Or rather, if we had Rust code, we'd have to like duplicate it in like [generator] machine code sort of. So these two are definitely intentional. We're like, number one, the runtime overhead is like very, very, very, very, very particular as to whatever strategy you want to choose in terms of making stuff safe. But then the constraint that we have of JIT code is modifying all this. It kind of constrains us as well. Like an example there is that like, so in WASM, you can have Globals, which are effectively, they're not really like Globals in Rust. You can't take the address of a Globals in Rust. Well, imagine a Globals in Rust, but you couldn't take the address of it. So you can only set it get it. You can't have those in WASM. And the way we even with those is that this VM context structure just kind of has an area of memory that stores every Globals. So a Globals get is just, you read the memory in a Globals set, it's just write the memory, but we have to statically know what the offset is. So like a Globals get of the fifth Globals is always at like byte offset two hundred twelve. And then similarly, if you import a Globals from someone else, you like load their pointer, get the pointer to the Globals and then like read and write from that sort of location. So anyway, this is sort of like an example of how the, because we have to interoperate with a JIT and we don't want that to be super, super complicated. And we can't bake like Rust's, old type system or stuff for representation inside there, but it has the sort of confirm and summarize.

13:51 We are pretty constrained in... No Other Choice

13:52 We... Increase Performance

13:52 We... No Other Choice

Interviewer

You, because you're working with a JIT, if you were to attempt to use unsafe in a way that corresponds with the whole idea of like minimal and independently reasonable, you'd run into issues where you'd have parts of your architecture that are duplicated. So that or along those lines, certain patterns that you'd want to avoid because they'd have some sort of architectural downside, whether that be just maintainability or performance. And so it's, you're, you seem like you're sort of stuck in this position where you're between a rock and a hard place where you have like a lot of unsafe code that might be difficult to reason about in isolation. But if you were to fix that problem, if you consider it a problem, then you would give yourself another architectural scenario that wouldn't necessarily be better. Do you feel that that's accurate or?

Participant

Yes. And I would actually refine that a little bit in the sense that we honestly don't have a huge amount of motivation to reduce the unsafe code in the last time. Now, not because it's maybe bad, it's more of like, we've sort of reached a local optimum where like performance is really good. A lot of us understand it and we don't have any like huge problems. We know with it. So given all those constraints and the fact that our embedding API is safe, so users don't have to worry about it. We're talking about purely an internal problem. That's only for us developers and all of us currently understand it and none of us know how to kind of like easily move it somewhere else. So it's sort of like balancing all those constraints that we just don't really have any motivation or it's, I would say we don't have motivation. It's just a lot of our motivations now are more like kind of in the list of priorities prioritizing like new features or bug fixes or other forms of fuzzing or things like that. And so this is where like, we also very much acknowledge that like our own understanding is obviously not sufficient. So we invest in areas such as fuzzing, such as Miri, such as, I think we're, we're trying to get an audit from, I think it's [redacted] or something like that. So there are other strategies where we're basically trying to do, which is not trying to make everything safe because we've sort of, we just don't know how to do it and try and invest in other places instead. Not sure.

13:53 A lot of u...
Audience-specific Rea
13:23...
Fuzzing
13:25...
Unsafe is Diffi...to U
13:24 we invest...

Interviewer
So this next question moves a bit more into particular type usage, as you mentioned. We already discussed Box a bit, but could you speak a bit more, you selected unsafe cell and cell as well as manually drop. I guess we should start with like the cell types, what you'd use those for in an unsafe context.

Participant
Sure. Um, unsafe cell is like, it's kind of a little loose and I think we're not really entirely certain we use correctly. I love the place, but like probably the one example is that we do have something which is stored in thread local storage. And so that once you access the local storage, you can really only get a shared reference because at any point in the stack, you can just reacquire it. So you can't get a mutable reference unless you're dynamically check. So we don't want to do have a dynamic dynamic check. So we just put it, we give you &T, but we want to mutate &T. And then sometimes we don't want to pay the overhead of like unsafe cell or RefCell, just because like a lot of these are, these can be hot paths. And so we use UnsafeCell instead. So some of these are like, we just inherently only have &T. We don't want to pay the cost of the RefCell. We can't statically sort of fall into the Cell concept. I'm like, that's a good example. I was just looking at this the other day. No, maybe that's not a, not a great example, but any case, that's, that's where the Cell types come up. We don't tend to use Cell a lot. It's mostly the other other, we mostly use a bunch of other types per se. So let me rephrase our usage of UnsafeCell is when we sort of inherently only have &T, or we're too scared to give out &T or to have &T. So maybe it's, maybe it's fine. Maybe it's not. We just kind of err on the safe side and we just say it's UnsafeCell. Another option is like, we use this to model some data with

13:54 Um, unsafe cell is like, it'...
RefCell<T>
UnsafeCell<T>
13:26 It's mostl...
Exposing a Safe API
...
UnsafeCell<T>

the JIT. So like, we make sure there's an UnsafeCell for data, which is mutated by both the JIT and by [tool] to make sure that we can't like accidentally hold an mut across a JIT call by accident, because the JIT could modify it and the Rust compiler doesn't know about that. Things like that.

13:55 An... UnsafeCell<T>

Interviewer
Gotcha. Okay. Um, I guess, is there a case where you use a Cell in place of an UnsafeCell? And would you have like a particular way?

Participant
We definitely use that wherever we can. Like that's just, I mean, why we, why use UnsafeCell, Cell suffices and that primarily works for like copy types. So if I have like a counter or something like that, we just use it up. We just use it, we just use a Cell. The UnsafeCell typically comes up with some sort of non-movable type. So like, for example, when you hit a trap, you will collect some data with the trap like some backtrace and some information like that. And we store that in a data structure, but the data structure is like, it literally only exists to store the trap and then not have a trap when it goes out. And so we store that as an UnsafeCell of a manually drop. And so that way, if you don't hit a trap, you don't pay the cost for running the destructor because otherwise you would have to like check an option, check the thing and like, as you go out, actually, linear phrase. This is the, this is the example I was trying to think of, which I just now thought of. So for example, whenever you start executing my assembly, our thread local state will store, this is the trap that happened. So you start executing my assembly and then during that, it's like going to do some stuff and then that'll catch the trap internally. And then when we exit, we're going to check and see like, did it return one or zero? And if it returns zero, I'm going to go load the trap because someone already filled my trap. So that sort of paradigm can't really be captured with a Cell or a RefCell, or it can be captured with a RefCell, but we don't want to pay the cost of every single time we invoke WebAssembly, we check the RefCell flag as we exit because it's got a destructor associated with the, because we know that the dynamic guarantee is zero is only returned if someone filled in that data. And so that's sort of like the UnsafeCell ManuallyDrop style stuff. That's what's one example of where we're using it there.

13:27... Cell<T>
UnsafeCell<T>

13:28 So like, for exa... Increase Performance
ManuallyDrop<T>
UnsafeCell<T>

13:29 So that sort... Contract or Invariant
Increase Performance
ManuallyDrop<T>
RefCell<T>
UnsafeCell<T>

Interviewer
Gotcha. And is that all?

Participant
Yeah

Interviewer
Gotcha, gotcha. And then also you seem to have very extensive FFI usage just based on what you selected for, what's interesting that it's, I guess, not from Rust into other languages necessarily, at least that you, you have more languages selected where you're calling Rust libraries

from those languages, it seems like, than the other way around.

Participant
Yes, I definitely have a lot of experience of calling at least C and C++ from Rust, but at least for wasn't times use case, we have a couple of embedding APIs, which is a lot of languages invoking Rust into the hood. Yeah, the weirdest one is in fuzzing, we call a OCaml rust because the spec interpreter for wasm is written in a OCaml. So they'll like run a wasm case in the spec interpreter and then we'll run in [tool], make sure the results are the same.

Interviewer
Okay, gotcha. That's not here on there. And then so the, the two, or I guess the three methods you've used to create bindings then were just manually running them using wasm-bindgen and then ocaml-interop, which I'm guessing that might be for the use case you just described, right?

Participant
Exactly.

Interviewer
Talk a bit more about each of these three methods of writing bindings and like, what, what's the usability experience, I guess, for the different languages under these cases and like, maybe what, what were some challenges you found, things that could be approved?

Participant
So the, the, the wasm-bindgen aspect was more so…we haven't actually used that for [tool]. So I've just, I've got a lot of experience with that. I know like that's used for, for a lot of like JavaScript and webby stuff, but for, for [tool] specifically, that's actually all been manual. It's all been, we wrote the CAPI manually, we wrote the C headers manually, and then we wrote all of the bindings and everything which manually, we don't actually have auto-directed, I think like that. And one of the primary reasons for that was basically just, that's what we saw was reasonable at the time, like the, the, the major thing we sort of wish we could automate is Rust source code to a C header file. We've definitely made mistakes there, which is that we had like the wrong size integer in the C header file versus the Rust code. So we, we don't do a great job of agreeing between our header files and our Rust code. So I wish we could do better there. And that's just one we haven't really invested any time to, to improve that. But otherwise for each native language, it tends to be that like the idiom we desire in each native language just really isn't well supported by some sort of automatic binding generation or automatic thing. So there's stuff we can do to improve it, but I don't know, let's, let's, I'll talk about Python as a specific concrete example, because I have, I've been doing a fair amount with that. That's right. So one of the, one of the first things that you might lean towards with the Python bindings is

13:31 we... Generation VS Validati

13:32 We've... Bindings are a Fiddly M

13:56 And th... Idiomatic FFI Encap

Python. This is just generally, I've got Rust code. I want to call it from Python. And it's great that we actually, our initial bindings were written in that. But one of the major drawbacks of that is you need to compile a version of your thing for each version of Python. Because it uses like, it's just the general C extension API and generally Python, you, I think Python six extensions, you always compile per Python version. So we're not Python experts and we don't really want to invest the time to do that. So I decided to not do that. I decided that the way we were going to do it was instead via the CFFI module. And so that's just because I didn't want to compile. I wanted to use the pre-compiled C API libraries from less than time. I didn't want to redo them, recompile them, make up CI for all that, have different architectures, different Python versions. It just seemed like a pain to do all that. Now I'm not learned about it. Maybe there's an easy little, you know, action or something like that to do it. But that was my art perspective. And then once we chose that path, and we decided to do that, we decided to once we chose that path, it ended up being that, like, there's not really any by automatic stuff from that point. So we ended up, I ended up just writing all the manual bindings on top of that. There, I wrote a tiny script to, like, go from the C header file to typed Python CFFI stuff. So, like, it would take the C header file and say that here's this function that takes two integers and returns a pointer or whatever. And then so given that, it was like kind of like a basis to build on. And then I built like a nice high level Python library, which is the same idea of, like, you can't cycle to use it, ideally.

13:57 I think Python six e...

Bindings are a Fiddly M

13:33 So we ended up...

Generation VS Validati

Interviewer

Okay. Along those lines, are there particular issues that you found in reconciling the assumptions that Rust makes about its memory model with the memory models use different languages? That's a very broad question, I know.

Participant

But I have the perfect answer for this. So just the, it's, we, like the only, okay. So if we rewind the clock, maybe four ish, no, three issues, this was a time when we did not have the current API that we have for the C API for [tool], nor the current Rust API. So this was like, our API had evolved to the point. It was just kind of whatever worked. It was just whatever was there, but it was clear that there was a lot of problems with it and we wanted to completely redo it. So not only the Rust API, but when I was designing the Rust API, I realized that it would have very large impacts on all other languages. So a great example is that, like, in Rust, you can trivially say, I'm going to return to you a thing, which just borrows my memory, and then you just can't access me while you're using that, like an iterator while you're iterating a vector, you can't push on it. Like, that's just trivial. Like, you expect to be able to do that, but this is not very easily mappable into a C API, or rather you could do it, but then you have to read all this documentation of like, please don't mutate the vector while you're iterating over it, and that that has to mirror to Python saying, and Python, please don't do it, because like in Python, you don't want to say, at least throw an exception or something like that. So it was, it actually was a very concerted effort to design the API in [tool] to be similar to what

13:34 in Rust, you can tri...

Different FFI Memory M

it would appear like in other languages. And basically what we settled on here is that, like, we're not going to have any borrows at all. We're just going to try and maintain everything as a nice own data structure. Some things are copyable around as is, and this is where like our store, like our store is the canonical, like you malloc some data, everything's inside there, and that's it. That's the only reference to all like, stuff. So it means that in Python, for example, if you have a store, then that's got some data and like the destructor for the store, is it freeze the store? And if you have a function from within the store, you can actually only interpret a function within the context of a store. And so this way, if you like mismatch, we'll just get a nice runtime error. It's very well defined. It's all every every function like takes a store with a function, like good stuff, or in Python, if you accidentally persist a function beyond the store, you can't call it because you need to pass a store in to call it. That's like proof you have the lifetimes and things like that. So basically, it's a very long winded way of saying we had to reconcile this that the API design layer was a very much, much higher layer than like kind of dealing with models or anything like that, because we wanted a nice kind of an a ergonomic API in all languages. Like, I think we've gotten like .NET, go Python and C++ of the main ones. And trying to cover all those at the same time, basically meant we just had to switch up some designs in the rest side of things. We didn't really like compromise per se was more of like trying to find a nice API and Rust, which additionally mapped all those. I don't know if this is that's exactly what you're asking for, but that's like the experience we had at least. So ensuring that you are not crafting an API where the types you're using will that are implicit in Rust will then turn into documentation in other languages that users have to rely on. So having a set of assumptions where you're just minimizing the burden on the user when they consume that API from another language.

13:35 And basic... Simple FFI

13:59 So basica... Idiomatic FFI Encapsul

13:58 So en... Simple FFI

Interviewer
Gotcha.

Participant
And so that it works out. It's one of those were like that worked out for the bulk of the API. Now, there's still some niche things where it's like iteration looks a little bit different and see that it doesn't Rust just because we want it to be safer by default, but in general, that was like the driving principle was don't try and like work around the languages or try and map rest of the languages. It's like try and find a nice intersection that works for everyone.

Interviewer
Gotcha. Gotcha. Yeah, that makes sense. And then, um, I guess this is another very broad question and describe a bug you faced that involved unsafe Rust. How did you find it, how did it manifest, um, how do you fix it and, uh,

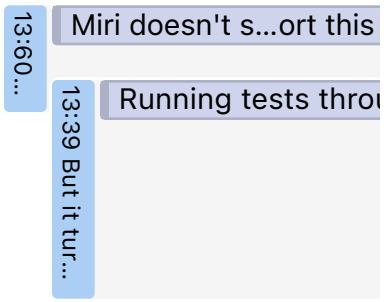
Participant
what I'll describe the CVE. We, so we had a lot of a number of CVEs on,

on, on [tool]. So I'll describe the most recent one because it's actually, it's actually pretty interesting with unsafe for us. So I was talking about this VM context earlier where this, this is a VM context, the way that it's actually laid out is we have some Rust, we have a Rust structure called an instance. And then when we allocate that, we actually were allocated with a VM context at the end. So the rust structure is static. It's like just some stuff we have in rust source code. And the VM context is a variable length data structure depending on the size of the module is like, if you have 10 globals, it's going to be smaller than if you have a hundred globals or 10 functions versus a hundred functions. So, um, those two are allocated back to back and then we have a like the instances of the front, the VM context is on the back. And then the VM context is like kind of a pointer in the middle of the structure. So when you have one pointer, you can infer the other, all that good stuff. So we had a runtime function in [tool] where it would grow a table. So like there's a, there's an instruction in [tool] table.grow. And so we had a little lib call, which like it went through and JIT-land. It gets a VM context. So what it would do is it, given a VM context pointer, I'm going to rewind it, get the instance pointer, the instance pointer, the instance pointer, then I'm going to like take the parameters to the instant, the table, grow a function call, go do some runtime stuff, grow a vector, all that good business. But then the last thing it was going to do was it takes the runtime state for the vector and it writes it to the VM context. This is where the VM context tracks like the length of the table and the base pointer of the table. So that way JITcode went to the table, get table set. It can just stay in JITcode and have to come out to rest. So that was a general operation. What we experienced though was a fuzzer found this is that on LLVM 16, which is on nightly or beta at this point, the write of the runtime data structure to the VM context was optimized away. So all of this was unsafe code because all of this is like we're just writing it a memory beyond our length and like it's in the VM context region. So there can't be anything safe about that. And so our write, which was crucial for correctness and just the validity was optimized out, which then meant we had use after frees, which then turned into a segfault in the fuzzer, which then turned out to be negative. That's its own CVE itself. So that was the context for everything. And the ingredients for what was going on here was that we had a function which the signature was that given ampersand instance, which is the beginning pointer and an offset from the VM context. So like take 10 bytes out, you can calculate the VM context pointer, go 10 bytes out and then return to the pointer and then we were right to it. But the problem here is that given the LLVM attributes involved, what this ended up being was a function that takes ampersand self that writes to ampersand self like to data that thinks that it's owned by ampersand self. So LVM thought that this pointer was read only and noalias, which meant that nothing else was like pointing to it's all pointers. Basically, it was basically a certain LLVM were not going to write through this pointer, but then we wrote to the pointer. So because it was undefined behavior, it just optimized it out, deleted it, all that good stuff. So anyway, all of that was based on safe, based on safe assumptions, based in like various abstractions we had internally. So the fix for all of this was actually literally changing ampersand ampersand self to ampersand mute self. That was literally all we had to do. The only change necessary. And that fix everything. Plumbing it, plumbing it. It's

13:36... No Other Choice
Writing beyond a boun

13:38 So LVM thought th... Borrowing or Provenar
Stacked/Tree...rrows Vi

very unsatisfying fix because it's like, OK, well, where else? If that's the one case of this, are there not more lying at me? So this was one of the main motivations that we had for running Miri. So we discovered we've never run our code in Miri because we've assumed that we are a JIT and Miri can't run JIT code, so therefore it's useless. But it turns out we have a lot of tests which are purely exercising the embedded API, so like create a table and then grow it and that's it. And all of that is like pure Rust code so can easily run at Miri. So our goal was to effectively run a couple of like some subset of our tests in Miri and sure enough, if you go run it on the vulnerable [tool], it instantly spots this bug. It says that like, oh, yeah, that right is invalid because you don't have right access to this pointer. So our fix was to that was the fix there, which is basically work through the errors in Miri and effectively change a little bit about these abstractions internally. Like I mentioned, it was ampersand self to a write. That actually is valid, but you have to model it in a very special way for Miri, which is all about like prominence and things like that and fun stuff.



Interviewer
Gotcha, gotcha. So that's a perfect segue then. So my last two questions relate to any of the development tools that you've used with Rust or any code bases that you interrupt with with Rust with. So you've you've mentioned that Miri has been very helpful. You also selected address sanitizer and then you mentioned a bit about fuzzing and you wrote that you've been using web fuzzer. Could you, I guess, so we've we've spoken about Miri a bit. Could you talk a bit more about where address sanitizer has been helpful, maybe where Miri hasn't been or like, how that combination would be more helpful than one of the other?

Participant
Definitely. So and Miri is a very recent, like this week, last week kind of development. We have not been running for a long time, but we have been running fuzzing for months or years at this point. Fuzzing is our number one tool for finding issues. And that's like we lean on that extremely heavily. If you're looking for a very simple comparison between ASAN and Miri, we can run JCO and ASAN can't run JCO and Miri. Like basically the costs associated in the runtime capabilities are very, very different there, but all that being said, ASAN, I might have misunderstood the question on the questionnaire, but ASAN is practically worthless to us because we never found any bugs in ASAN. Like ASAN is generally intended for like C and C. ASAN is typically intended for like C and C plus plus. So if you just have memory pointers everywhere, everything's unsafe. Who knows what? So for us, we actually have never really had any issues with ASAN. We've never had use after frees.. We've never really had dangling pointers in that regard. Not to say, I mean, they have existed occasionally, but like ASAN has been relative to the cost, which ASAN gives you like a 10X-ish cost when we fuzz. It's not really worth it per se, but fuzzing, however, fuzzing is an absolute godsend. We, I mean, basically, we put a huge amount of time and effort into maintaining our fuzzers, keeping them running well, fixing bugs in the fuzzers. And this is where [tool] is a project that runs on OSSFuzz, which is a service provided by Google to just run open source software on their on their

infrastructure for free, which is an absolute godsend for us as well. We get, I think, hundreds of hours a day of fuzzing runtime. Just they just keep it to us and they've been doing this for years now at this point. It's amazing. So the ASAN aspect is not really particularly useful for us. And it kind of goes back to like, despite having a lot of unsafe code in [tool], a huge amount of it is still safe code. And so ASAN, like when you're like, there's not really any reason to run ASAN for safe rust code. Same reason if you have pure safe code, why I really want myriad of it? It's just all safe code anyway. And so we found Miri useful for the unsafe code that kind of like the stuff that ASAN is not catching, ASAN is catching it like after it happens, whereas Miri is catching it sort of before it happens in a sense. But anyway, so that's like ASAN. But fuzzing, primarily the concerns for us have been coverage and fuzzing of getting interesting test cases. So generating interesting test cases for us, interesting Wasm modules, interesting combinations of API functions. That's always been a very big difficulty for us to try and write good fuzzers for that. But otherwise, that's the number one by far away tool that we use for our correctness is just purely fuzzing.

Interviewer
Gotcha. And there's an interaction, I guess, between the fuzzing and Miri that you mentioned, where the bug where you were writing to something access from and self that was discovered, it seems, first by a fuzzer in the way that you described it, and then you use Miri and confirmed that that was a borrowing violation of Miri. Have there been other cases like that where there's been like a synergy between the two tools?

Participant
107 I guess since Miri is fairly new, that might not be. Yeah, that's the first one. Most of our fuzzing issues tend to be a lot of historical ones have been actually garbage collection related. So this is where extra ref is a data type in WebAssembly, which has just like a data from the host and it can be garbage collected and things like that. So we have a various a scheme for garbage collection and we've had a lot of issues just correctness of the compiler of like it doesn't generate stack maps. It generates incorrect stack maps. It forgets an extra ref here and there. That's just sort of a couple of issues around in the past. So we just kind of had those problems occasionally from time to time. And we've also had some code generator bugs. So like WebAssembly relatively recently gained a SIMD instruction set. So just 128 bit vectors and various data types over them. We added a differential fuzzer, which will take code, execute it in the spec interpreter or something else and [tool] verify has the same results, which then added support for our fuzz case generation to generate stuff with SIMD instructions and then we just ran that through the fuzzer and it occasionally finds issues of like this lowering is a little buggy. Like when you have this kind of weird shape of module, it just has a compiler panic or it produces the wrong results. Things like that. So I guess in some general, no, a lot of most fuzzing bugs have been we find the fuzz bug and then we just sort of trace that directly back to a bug in the code and we don't really need any extra tools to help figure out where.

13:63 Most of our fuzzing is...
Fuzzing
Logical Error
Requirements Bug

13:42 it o...
Logical Error
Requirements Bug

112

113 **Interviewer**
And then with the Miri configuration, since you're sort of limited by the support that Miri has for code in various formats, is there a mechanism you've been using to, like how do you determine which parts of your code base that you want to write tests for in order to have your mirror be effective on them? Like is, are you just filtering from your current test suite, the ones that you can handle or you, gosh, is it, is there a different method?

114

115 **Participant**
Yeah, no, I, again, Miri is very new, but the way it's done so far is I ran the entire test suite in Miri that took like 10 hours and then I filtered all the tests that did transmutes from pointers to, like from a Rust compiled function pointer to a function and then that left me a set of tests, which is this basically all of these run in Miri. So we filtered that down to run in Miri. So one thing I've thought of though, in terms of tools in conjunction with fuzzing RR, I don't know if, have you heard from me with RR as a, as a debugger? Oh, let me tell you about RR. So RR is a, it is sort of a GDB front, it's sort of not. It's a tool from Mozilla, but it supports reverse execution. So the idea is you record a trace and it will, then you can replay that trace and you can go backwards and forwards inside of a debugger. And for debugging most fuzzing related crashes, that is an absolute godsend. Cause a lot of times it's difficult to reproduce things outside of the fuzzer. I could tend to need like really funky setups. You tend to need like kind of big, just embeddings. It's just not, it's, and like you got to, you got to match all the settings exactly. So it's a lot easier if you just run the fuzzer and then debug it within the fuzzer. So RR has been, we have so many times debugged and we're like in JIT code. So we don't have any symbols or anything like that, but we can very easily stare at this simply walk back, 10 instructions, walk back, like I kind of just go back and forth with instructions, and that has helped us debug so many issues. That's kind of specific to us of like, I mean, it's, it's all JIT code related, but RR and debugging like GDB is only a bit so helpful because like, I mean, Dwarf and Rust is not always the best thing or debugging for, but R, specifically being able to stop, go back, replay it and do that continuously has been extremely helpful.

13:43 Miri is ver...
Miri is slow
13:64 So one thing I'v...
Debugger
13:65 That's...
Compilation & Interpre
Debugger

116

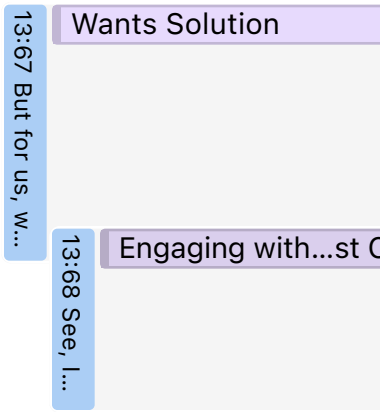
117 **Interviewer**
Gotcha. Gotcha. And then are there any particular problems that you face when ready and safe or verifying its correctness that your current tools can't solve for you?

118

119 **Participant**
Hmm. Well, I mean, the biggest one facing us right now is we would love to run Miri, but actually run the Tbizom test suite. Like, WASM has spec tests, we would love to run the spec tests in Miri, but we don't know how to do that right now. Like our closest approximation is going to be we turn on the [generator] interpreter and then we interpret everything in Miri, but that would probably take about 20 hours of execution. So that's probably

13:44 We...
Miri doesn't s...ort this
13:66...
Miri is slow

the main one. Well, that's a big one. Let me see. Where else have things fallen down? Honestly, there's a lot we could say about fuzzing as well. Fuzzing is good, but it is not an absolute panacea. And there's a lot of things we have done to sort of like improve our fuzzing, but as we have a long way to go as well. And this isn't really specific to like Rust per se, but there's just various strategies on fuzzing that a lot of stuff, like custom counters is a great example of like a lot of cover, a lot of fuzzing right now is based on coverage. So if I run this test case and it touches more codes, the codes touches more paths in the code base, it's considered a good fuzz test case. But for us, we're much more interested in kind of like interesting shapes of code. Like, did you consume more stack frames? Did you consume more heap? Did you have like more instructions in various weird shapes here and there? And it's been very difficult. Like a lot of the coverage based heuristics or maintaining test cases have not been sort of like, we don't think they're necessarily perfect for us. Um, there's that. See, I don't know. I mean, we would certainly love for someone to walk in and tell us that like our architecture for [tool] could be so much simpler if we did this one thing, which has like no runtime overhead, so much safer. I don't have no one's ever come forward with that. I'm not sure it can happen, but that's what I have on the top of my head in terms of tools we wish we had or things we wish we could do.



Interviewer
Well, you're in luck because I have just that.

Participant
If only.

Interviewer
Yeah, I can definitely relate it to being in that situation recently, architecturally. Yeah.