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

Participant
Okay. Well, I first heard about Rust in a conference back in 2017, and at the time I was also in academia. We were investigating some issues with state management and operating systems and just general operating systems principles. Once I learned about Rust, the ownership system and kind of like the whole affine type system based safety and the model surrounding that attracted us for a variety of research purposes. So I think that's why we started using it. As we used it more, we used it to create a new OS from scratch. So as you can imagine, lots of unsafe opportunities. As we used it more, we became more enamored with the ownership system and the general benefits of linear typing. So I think we've explored a great deal of Rust, not just Rust, but the power of language as it can pertain to improving the safety and reliability of operating systems.

12:1 O... Memory Safety

12:... Operating & E...dded S

Interviewer
Gotcha. Could you be a bit more specific about the state's issues that you were encountering and maybe what aspects of Rust seemed appealing in light of those issues?

Participant
Sure. Yeah. So this is an old research topic that I haven't really stayed up to date on over the last few years, but the initial idea was something along the lines of it's difficult to determine how state moves and propagates throughout subsystems in an operating system. And ownership seemed, at least on the surface, a better way to clarify that or make it more clearly defined. So opening it up to some static analysis tools, because of course ownership and borrowing can typically be known, as you know, statically in most cases. So does that answer it? I don't know.

12:17 it's... Memory Safety

Interviewer
Yeah, no, that's perfect. So then what do you use unsafe Rust for in the Rust development that you do?

Participant
Lots of things. Yeah. So one of the goals of my project, so by the way, the project that I'm working on is [os]. Like I said, it's an OS written from scratch in Rust. I do a variety of other projects, but this is probably the one with the most unsafe. So inherently an operating system has to talk to hardware, right? So like with a Rust standard library, something you have these nice clean abstractions that kind of wrap around some raw primitive operation at the lowest level in a standard library. It's typically something like a system call, right? So this reduces things down to the low level types. And as you probably know, there's no real way to avoid

using those within the kernel itself, right? When you're talking about the interface between low level software and the hardware, like the ISA or a device, same thing, right? You got to use raw primitive types and that necessitates, you know, unsafety in many cases. So to be more specific, you know, in operating a system, you know, you need to communicate with the CPU model specific registers on X86. You know, ARM has a similar notion of these various hardware registers and inherently these are unsafe to access because they can have loads of side effects that could completely change the behavior of the system, be it things related to how virtual memory is managed or things related to, you know, bootstrap of the system when you're taking over from a bootloader, right? So, and then on the things like device access, you know, it's the same thing. Ideally, we try to mask all of this unsafety with safer, safe, higher level interfaces, but at some point, right, like in the house somewhere, you need to descend into the low level raw primitive access, you know, like writing bytes to a particular region of some MMIO register. It's just, it's unavoidable to a certain extent.

12:3 So to be more specific, you know, in...
No Other Choice
12:...
Exposing a Safe AP
12:...
Interacting with ha

Interviewer
Gotcha. So, would you say that the use of your, like of unsafe in this project is generally by necessity in these cases where you're accessing lower level hardware and then you're relying on, relying on safe abstractions wherever you can, or are there cases where you would choose to use unsafe for something over a safe abstraction maybe to improve performance?

Participant
Ah, yes. It's the vast majority in my specific case is the former where, so part of, I think part of like the charter of this project is to see to what, like, what's the fullest extent to which we can avoid unsafety, right? Yeah, of course, there are many, many cases where we cannot avoid it. There are definitely some cases where I have considered using unsafety either because it's easier than developing a safe abstraction or because of potential performance issues. I don't know. Okay, actually I can't think of one case where we had intentionally chosen to do something unsafe and that was a heap allocator implementation. So the idea is you have some regions of memory and this is before you have a heap since you are the heap implementation and you need to maintain some metadata about what is allocated and what is free, right? That's the most generally abstract way I can present it. So in order to do that, you need space to store that metadata. So you typically store it within the actual memory
11 that has been allocated for a particular purpose, right? You allocate, you know, let's say you allocate eight kilobytes, right? So some tiny part of that eight kilobytes is used for the metadata, right? There is apparently no way to represent this in, say, Rust, right? This is in effect a self-referential data structure, right? So that's, as you probably know, that's a very hot topic in Rust and how to represent them safely is kind of an ongoing issue. So that's one particular place where the overhead of, there's two problems. One is a sort of a cyclic dependency between needing to allocate something and not having an allocator, but you are in the allocator itself. So you can't allocate something inside of the heap

12:66...
Preference for Safety
12:5 There are...
Easier or More Ergono
Increase Performance
12:6 O...
Custom Allocator
12:7 So you typi...
Multiple / cyclic aliasir
12:67 Thi...
Shifting Ground

15 allocator because you're the one providing that allocation, right? So there's a little bit of a cyclic dependency there that needs to be broken. And typically the way that we do that is we have a little bit of a static memory reserved for this, you know, a few initial allocations to kind of bootstrap the allocator, right? Very common approach. So that's one aspect of, that's one reason why it's hard to represent this sort of behavior with fully safe code. So you could do it, right? But that would be some performance overhead, right? So you'd have to, and it's also limiting in the very nature of the way that you could use it. So for example, if I only reserve, you know, I don't know, 128 bytes or something for this statically, then I can't, you know, if I change my system to where I need to allocate so many more regions of something before the heap allocator is available, then I can no longer do it without making changes to the code. So it makes it more of a brittle interface. That's one problem. I was going to say something else about the heap, but I forgot what it was. Oh, all right. So anyway, we had, so we had tried, we had like three heap implementations. One's like completely unsafe, one's moderately unsafe with a safe interface, and one's fully safe and a fully safe one was significantly slower, right? Just because it was mostly wasteful of memory and then we're passing around all these reference types to make sure that things last as long as they should, so like arcs everywhere. Not great, right? We have, in these case, we have a variety of different types, high level types that represent a safely allocated and, you know, always actively mapped to real existing physical memory, regions of virtual memory that you can access. There's some types that represent that. So we attempted to use those in the heap themselves. So it's basically like we're allocating memory and tracking those applications using heap allocated data structures, like a vector or a linked list or something like that. And then underneath that is the actual heap memory that's stored in that is then backed by the underlying memory that was allocated. So it's another weird cycle, right? So because of that, you know, to represent something like this, you're using like kind of a graph of Rc/Arc and weaks everywhere, and it's slower than just assuming, you know, with unsafety and, you know, or requiring rather the implementer of everything internal to that crate to make sure that it's done correctly. You're not, you know, freeing something before you're actually done using it. But yeah, in that case, we settled, I mean, we have all three available still, but we settled on a choice that was, you know, partially unsafe but still provides a somewhat safe interface. And of course, the very nature of the global allocator interface in Rust, the trait for the allocator itself provides two unsafe functions. So you have to use unsafe to implement them, because Rust wants to be sure that you're implementing them correctly. Yeah, yeah. I know that was a lot, but I'm happy to clarify or elaborate if you need more detail.

16

17 **Interviewer**
Yeah, yeah, I guess just to backtrack a bit. So the main problem is you are allocating something, but you're doing it within the allocator. So you're using yourself to allocate something, sort of, and because you have a cyclic dependency.

18

12:8 So t...
Increase Performance
12:9 So you'd have...
Easier or More Erg
12:10 Jus...
Arc<T>/Rc<T>
12:68 So we attem...
Multiple / cyclic aliasi
12:11 So bec...
Arc<T>/Rc<T>
12:...
Exposing a Safe API

19 **Participant**
Yeah, that's right. If you can, I don't know how familiar you are with allocator algorithms or if you are at all, but a common one is like a buddy allocator or what I mentioned, you know, where you have the point is like you need to have some metadata about what is allocated, right? You have a chunk of memory, say 8K, and maybe you want to allocate just one 8-byte integer. When you do that, you need to maintain some metadata about the fact that those 8 bytes, wherever they are in that range of 8K, have been allocated and can't be reallocated by another future allocation until it gets dropped, right? That's inherently the purpose of an allocator. So you need to store that somewhere, but until you have an allocator up and running, you don't have any memory to store it, right? So you can't like use a vector, right, of ranges of virtual addresses that have been allocated, for example, because you need allocation for that. Yeah, this is basically, it's not truly self-referential, but it's sort of self-referential. So there's no way to do that single. Isn't like, so I guess you would need to, so you're storing this metadata within each existing allocation in some sort of chunks. So it isn't that you need to like allocate for the metadata and then allocate for the allocation of the metadata and then sort of infinitely. Well, you could do that. You need to make sure that you have enough memory ahead of time already pre-allocated. And the thing is, it gets complicated to make sure that you don't cause like basically some kind of recursion or maybe sort of a live lock scenario where you're allocating memory to store the metadata that was needed for a requested allocation.

20

21 **Interviewer**
Gotcha. Yeah. Wow. That does sound very challenging.

22

23 **Participant**
I'm sorry, I think you're, I'm not sure if it's my connection or yours, but I think I didn't quite hear the, the last, it told me that I'm not sure how I can. I can switch computers if this becomes a problem. But anyway, so what I was saying, I think before it froze, yeah, right. So there's, there's a cycle, but if you have, if you introduce unsafety, you're basically, you know, of course, as you know, promise into the compiler, that the place that you're writing this metadata to has already been allocated. It's a real memory that exists, right? It's not going to cause like a page fault or user space aside fault. And you know, it's, it's memory that you own and is available for your usage, right? There's no real way to communicate that to the compiler.

12:71 So there's, there...

Contract or Invariant

24

25 **Interviewer**
Gotcha

26

27 **Participant**
So yeah, the allocator is one example where we experiment different, Oh, damn it again. Okay. Sorry. Let me, I can try my other laptop and join from there.

Interviewer
We could also just try turning video off. Maybe that will help with bandwidth.

Participant
Sure. I think that this is, so I'm on like an older tablet. I think the issue is that it's connecting to my 2.4 gigahertz network.

Interviewer
Oh, gotcha. Yeah, that's fine.

Participant
Yeah. But, but go ahead and ask the question as long as you can hear and see me, you know, I think it's okay.

Interviewer
Okay. Gotcha. Yeah.

Participant
I think I got that.

Interviewer
So I guess with, with the abstraction you settled on, how did you communicate, like, I guess the, the point of having like that safe abstraction is so that you like when you're using that API, you as a, like a consumer don't have to worry about enforcing certain invariants because they're just expressed by the type system at that point.

Participant
Right. So like that's the journey. Yeah. That's kind of what you call unsafe interfaces. You want to wrap them in something safe such that the invariants are checked before you enter an unsafe region of code, uh, specifically for like the heap. I mean, the, the very interface for the heap is unsafe, but then it's wrapped around higher level types like Box, arc, vec, which have safe constructors. So assuming that you implemented your heap correctly, those will always work.

12:13 I m...
Arc<T>/Rc<T>
Box<T>

Interviewer
Gotcha. So it's really that you had this underlying heap implementation, but then you were able to rely on Rust's safe abstractions just by swapping in your implementation of the heap for what the default would be.

Participant
Yeah. Yeah. So in an operating system, like this is a freestanding

environment, so there is no default heap. A default heap like jemalloc or whatever else would rely on an underlying operating system provided memory manager, but we don't have that. We are the ones providing that. So I can actually speak about our memory management subsystem itself, which underlies the heap. And that is also riddled with unsafety. Again, we try to make it as safe as possible at certain levels of the interface. So you're talking about mapping a virtual page to a physical frame. That sort of thing, inherently unsafe, because you are writing physical page table entries that the hardware MMU can then know about and understand, and you've got to make sure that you're doing that correctly, that you're calculating the offsets that go into each one of these page table entries correctly. It's about making sure the math is right. So then we have a bunch of types that atop the underlying unsafe virtual memory implementation, which try to, I mean, in my opinion, they succeed, but the goal of them is to strive to remove unsafety for anybody accessing or using a virtual memory subsystem externally. So foreign crates accessing virtual memory. So yeah, we have a myriad of different types. I don't know if that's relevant to this study, but I'm happy to discuss.

12:... Extensive, Non-local U

12:14 That sort... Contract or Invariant
No Other Choice

12:16 So the... "Safe" API
Exposing a Safe A
Tacit Knowledge

Interviewer
Oh, no. That definitely is. I'm actually, yeah, I'm curious in particular, like for both the memory management system and for the allocator, like what some of the invariants were, especially if they're things that are not like the immediate ones, like, oh, maybe just not having an all pointer. And how did you encode those as types in this safe interface? Yeah, hearing more about that would be interesting.

Participant
Oh, yeah. Yeah…the main idea is that you want to abide by memory safety rules when you have a region of memory that has been mapped and allocated and then mapped, right? So in [os], we have a bunch of sort of unique rules where, you know, a page needs to be globally unique, a frame needs to be globally unique. So you don't have multiple pages aliasing the same frame, mapping the same frame, right? Because that would violate Rust's aliasing XOR mutability guarantees, right? That's one of them. Another thing, so what we try to do is to use the Rust type system and, you know, the power of affine types and the borrow checker to kind of integrate what it means to be safely accessing memory into the compiler and express it in terms that the compiler can understand. So another example is if you allocate a, sorry, if you like map a region of memory, right, let's say it's a region that represents some DMA for some device, something like that, or like a register for a device, right? Like a hardware timer device has three registers that you can read the period, the counter and set up an interrupt or something, right? You want to make sure that you're representing those correctly, but it's not necessarily unsafe if you don't represent them correctly. It's just, you know, wrong, right? So having something be wrong is not unsafe necessarily, but what we want to do with that is to make sure that, like, you know, the driver, for example, the timer device driver, whatever it is, the network device driver that is accessing those regions of memory is guaranteed that, you know, for so long as it holds on to this, you know, particular type that we 47

12:19 in [os],... 12:25... Contract or Invaria

12:75 An... Leveraging th...orrow C

12:76 So another exampl... Contract or Invaria created, it can be guaranteed that that memory region will be actively mapped. It won't be unmapped by somebody else. It's not accessible by anybody else. If they mapped it, if they try to access mutably, we check and guarantee that it was mapped in a writable way so you don't get like unexpected page faults. And then the only way to actually access that memory is to borrow it as an instance of a specific type. There are a lot of, you know, restrictions on what that type can be, you know, given by various trait bounds. But we check, you know, the size of that type. So this is, I'm talking about like overlaying a struct onto an untyped region of memory. So we have a memory, let's say you have, you know, a representation of a timer and it has three, three MMIO registers. And you want to make sure that, you know, when you're accessing those MMIO registers, someone's not going to yank out the memory underneath you, or you're not going to allow a reference to one of those MMIO registers to persist longer than that of the actual underlying mapping, which would be a problem. So that's kind of a lot of what we focus on in the memory management subsystem. So inherently it's unsafe to map something, but soon as interfaces, you can be guaranteed that, ideally, right, you can be guaranteed that, you know, you're not going to encounter any problems like, you know, unexpected page faults due to non-writable access or, you know, accessing it accidentally after it's already been unmapped. Again, I can provide more details on how that works, but that's one of the ways we, you know, abstract away the unsafety in virtual memory management.

12:18 It's not acce...
12:23...
Runtime Assertion
12:26...
...
Runtime Assert
12:77 So we have a...
Contract or Invariant

51 **Interviewer**
No, that was awesome. Yeah, there were two concepts I sort of want to clarify from that. The first is, would it be accurate to say, so the properties of uniqueness that you're trying to prove here are something that Rust is sort of designed around, but it seems like something that you'd want anyway if you were implementing this in any language. Is that correct?

54

55 **Participant**
Well, so there's properties of uniqueness at multiple levels. There's like a unique type, and then there's like a global uniqueness of a particular instance of that type, right? So what I mean by that is like, let's say we have a type that represents a range of frames that has been allocated from the frame allocator, right? This is, you know, I want this to be globally unique such that two separate entities cannot allocate and use the same physical memory frame at the same time for two different purposes, right? That would violate aliasing extra mutably. It leads to tons of data races. You know, another example is on the side of virtual page operating systems. I don't want to allow multiple pages to all map to one same frame. I'm having more internet problems, it says. I don't know if you caught that. Right, so if you have a virtual address, like a page, right? That's a pointer, so you cannot have multiple pointers that could point to the same object concurrently, you know, outside of the bounds of something like Arc that the language already knows about, right? So if you did that, you would be able to create multiple independently owned objects that are seemingly completely separate from one another, but they actually refer to the same underlying data. That violates memory

12:42 a t...
Contract or Invariant
12:43 I don't want to allo...
Contract or Invariant
12:98...
Arc<T>/Rc<T>

safety.

⋮ Contract or Invariant

Interviewer
Yeah, so it's really that these are, and these invariants and properties that you're talking about, like, you're phrasing them in terms of the restrictions that Rust has, but these seems like things that you'd want anyway if you were to hypothetically implement the same thing and, like, see for some reason. So it's just that, like, would you say it's that you're, this problem is, like, uniquely suited to Rust, then, in that the restrictions you have are, like, already set up to be established by Rust type system, or is it more that there are certain things you need to do just to satisfy Rust?

Participant
So you don't have to do any of this, but I think in order to provide a truly bulletproof safe API around this inherently unsafe procedure of mapping and handling memory, it's nice to assert that these invariants are actually proven, enforce these invariants, rather. If you don't, you're offloading that onto somebody else. So that's what I would say many existing systems do written in unsafe languages. I mean, if you think about Linux or something, these guarantees don't exist. It's on the user. Like, for example, the user could be the Rust standard library. It's implementing safe abstractions around these things. But let's say if you have some shared memory, memory shared between multiple different processes, separate instances of Rust, there's no way for it to know about that. So that's outside of the scope of Rust. So it's not necessarily that all of these concerns are specific to Rust, but specifically for our project, one of the goals was to try to implement these various subsystems in a way that the Rust compiler could help us in forcing some of these invariants. For example, that's why we only allow a user, for example, a device driver, someone who has access to one of these mapped memory regions, to borrow it. You can't create an owned instance of a struct that is actually backed by some underlying memory because then you've divorced them. You've separated them such that there's no relationship between the lifetime of that instance of the struct and the lifetime of the underlying memory mapping. So that's one of the clever ways in which we use Rust. But it's also theoretically somewhat limiting. So yeah, I would say that's a great example of taking something that's unsafe inherently and then kind of wrapping it up in a safe abstraction, using the power of Rust to do the borrow checking for us.

12:27 I th...

12:78 For example, that's why we o...
Contract or Invariant
Leveraging th...orrow

Interviewer
Yeah, gotcha. And then you also mentioned that there are certain things that are bad, but that are not unsafe accessing a hardware timer. Is it the ...

Participant
Yeah, so let me give you an example. Like if you, let's say you have the layout in memory of those, let's use a very simple device like three register timer or something. Let's say you've made a math mistake. You flipped one of them. You put the counter register at offset eight and the

12:39 Ye...
Contract or Invariant
Logical Error
Requirements Bug

frequency register at offset zero and it should be the other way around or something. You're not causing unsafety, you're just incorrect. So for that case, we can expose those members of those structs, accessors to them in a fully safe API. There's no problem with that. However, if those functions are doing something that could then break another part of the system, like some side effects, for example, accessing configuration registers that dictate how virtual memory works, like the size on ARM, for example, you can set certain system registers that dictate how many bits are used for a virtual address. Well, if we allow someone to change that willy nilly, like all of the other assumptions for the whole memory subsystem, just go out the window. So that is inherently unsafe and there's no way to make that safe. So that's, you know, restricted behind public private APIs or it's, you know, if you wanted to expose it, which we don't, but you would have to do so unsafely.

12:... Contract or Invariant
Logical Error
Requirements Bug
12:41... Interacting with hardv
12:79... Preference for Safety

Interviewer
Generally, black and white is the spectrum of certain things that are just clearly where we're at.

Participant
I don't think it's black and white. And it can be, in cases, it can be quite hard to determine whether or not it's actually safe to expose a safe API. Sorry, I should say whether or not it's sound to expose a safe API to one of these, you know, underlying registers, right? Another example is like, if you're just reading the state of like some interrupt controllers, right? There's nothing wrong with like, you know, getting statistical information, right? That's fine. It's okay for like an unprivileged user to do that or an unprivileged foreign crate. No problem. But in some cases, like reading a status register, it may also clear it or have some other side effect, right? So should that be safe? It depends on the context of how it's being used, right? So of course, you know, you typically just err towards the side of unsafely being somewhat viral and that it affects all of the interfaces surrounding it. Only, you know, a good application designer, and I'm sure every developer will tell you this, anyone that's worth their salt will tell you that you shouldn't just, you know, wrap, uh, safe APIs around a bunch of unsafe stuff without doing a lot of due diligence. So, you know, for that reason, we have also embarked upon some formal verification ventures with formal verification tools that can help verify rust code with like pre and post conditions, but that's not always possible. I can give you some more examples of things like, uh, that we could never prove that are unsafe, like unwinding, right? It's just impossible. There's a bunch of examples, but yeah, I think maybe the device IO and access and system registers are maybe the easier ones to understand.

12:36 I don't think it's black an... When to Encapsulate?
12:... Interacting with ha
12:80 Onl... When to Encapsulate
12:20 So,... Formal verification

Interviewer
63 Okay. So I guess which, uh, which verification tools have you tried? This is a separate project that I'm not strictly leading.

70

71 **Participant**
I'm involved in it a little bit, but as far as I know, we're using Prusti, Prusti,

... Prusti

P-Rust-I, yeah. Yeah. And there are some other ones, the Kani verifier looked at, but haven't ultimately experimented with. But I'm also part of the formal methods working group. So we do a little bit discussion of this, but I am not a verification expert, but like some of, you know, some of the things like, you know, these properties in the virtual memory subsystem about like global uniqueness of a given page or friend, you know, ensuring that you're mapping, you're actually creating the mapping correctly in the hardware page tables from a given page to a given frame. Like these are things that we are trying to use Prusti to verify, right? Because I assert that they're sound and they're done correctly, but you know, that's me just manual inspection. So it's, it could be, you know, more formal.

12:... Kani

12:82 some of,... Contract or Invariant | Formal verification

12:83... Tacit Knowledge

Interviewer
Gotcha. So what are, what are some other examples of things that you feel are impossible to verify like the, um, second one thing that you mentioned?

Participant
Yeah. Yeah, quite a few. So unwinding, um, I, I don't, I don't know how familiar folks are with this, but it's, you know, basically like, uh, a stack back trace or running, uh, clean up the disruptors after panic, right? Kept in an exception, catching a panic, right? That's all, all implemented by unwind. So in the dwarf, uh, unwinding table format, which is like the one that everybody uses except for windows. Uh, so basically the way that works is you have a series of tables that tell you, if you had a, uh, an exception of panic at this instruction pointer, here are the things that are on the stack that need to be unwound and their drop handlers need to be ran. Right? So it tells you, Oh, Hey, here's the address of the drop handler that you need to run. If you had a panic in this range of instruction pointers, right? That's just an address. I just have to trust it. I have to jump to that address and I really can't do anything to verify it other than, you know, because I know about memory mappings, I can check that it's like executable and it exists, but I don't know what code, right? There's no way to like frantically know that at that location in memory, those instructions correspond to a drop handler for this type, right? That information is lost during compilation. I'm not saying that it should be there. I'm just giving you an example of something that's fundamentally unsafe and unavoidably. So, yeah. And you know, the unwinding flow jumps around a lot. So you, you jump back to that, uh, you jump to that landing pad, which is the, you know, where the drop handler is located, you run it, and then it should return back not to where you called it from, but jump back to another point. Right? So then you, you have another entry point that is, you know, it's past an opaque pointer. And I, I don't know who called that. I hope it's the landing pad that just finished running that I originally jumped to from the unwinder, but I don't know.

12:29 That's just an addr... Difficult to Encapsulat | No Other Choice

12:... Opaque Pointer | Transmute

Interviewer
71 Yeah. Right.

Participant
So that's another unsafe entry point, you know, and there's nothing I can do about that. I just have to trust that I parsed all of the unwinding information.

12:84... Tacit Knowledge

Interviewer
Right.

Participant
And the control flow and as I expected.

Interviewer
Yeah.

Participant
But, you know, this is okay because, you know, assuming that you got all that right, the there's no, like, there's no way to access this, right? Like from a user's perspective or a foreign crate, you can't like jump directly to a drop handler for a type, unless you're doing something inherently unsafe. You have to use unsafe code for that. So nobody would do that.

12:85 But, y... It's not a problem

Interviewer
Right.

Participant
It's, you know, it's similar, like you can't randomly like begin, you can't randomly enter that unwinding resume point, that, that entry point for resuming and unwinding operation, unless you just like, you know, cast a pointer to a function and call it and that's requires unsafety.

Interviewer
Right.

Participant
So these are generally things that are internal and they happen, you know, after a panic occur for example, right. To recover from it. So, yeah, I mean, but implementing the unwinder itself, right. That is inherently unsafe.

12:30... No Other Choice

Interviewer
Gotcha. And then, yeah.

Participant
Oh, go ahead.

Interviewer

I guess, my next question, you selected using almost most of the types that I listed in the survey in unsafe contexts. So I think I'll ask this a bit more broadly then.

Participant
Oh yeah. Just remind me, I've been on a work trip for the last two weeks. So I completely lost all context.

Interviewer
Yeah. So I guess the survey question was just which of the following reference and memory container types have you converted to raw pointers or used to contain raw pointers?

Participant
Okay.

Interviewer
You selected pretty much everything. So I was wondering just more broadly, are there particular challenges that you've had dealing with doing that conversion safely?

Participant
Yes, I would say so. So actually in the unwinder, there's an example of that. Like I said, the interface is weird. The control flow is weird where you jump to a drop handler and then it doesn't return to you. It jumps to another separate entry point. So because it's a C style of API, you just get one integer value. So we use that to hold a pointer to the context of the general unwinding flow at that moment. Right. So of course, I can't just like, you know, get a pointer to something that's currently on the stack. It may not exist. So I got to Box it up, then use it to raw and then pass that around in the context. And then to access it, I didn't need to unsafely reconstitute it into a Box from raw. So you know, it's just me trusting that the control flow is correct and nobody else tampered with that pointer or the contents of that pointer. That one is not particularly complicated, assuming everything goes right. But there were some other cases where some of my collaborators were misusing Box from raw for a pointer that did not originally come from a Box. It sort of worked, but I feel like it would have had major problems had we done anything complicated with it. So misuse of the from raw thing in an unsound manner, because it's very easy to mistake that mistake a Box for a generic pointer rather than something that was specifically allocated on the heap and came from a previous Box.

12:... Simple FFI

12:31 o I got... Box<T>

12:32 But there were... Box<T>
Box<T> Bug
From Raw, Wrong Poin

Interviewer
Right.

Participant
101 The other cases where we have this sort of thing is where we're passing

... Interacting with hardw

something to hardware. Like I have a virtual address or I have a piece of hardware where I need to write a virtual address into a register.

12:... Interacting with hardw

Interviewer
Right.

Participant
And of course, I got to get an unsafe. I have to get a pointer. Now it's not unsafe to obtain a pointer, as you know.

12:... No Other Choice

Interviewer
Right.

Participant
But then you need to make sure that it's sort of like any FFI interface. In the FFI, you have to make sure that thing actually lasts for however long the hardware is using that pointer. So typically we'll Box it up, put it in a struct that is tied to some other struct that represents a capability to access that piece of hardware to ensure that it lasts long enough.

12:89... Contract or Invariant

12:33... Box<T>

Interviewer
Right.

Participant
And then when it's shared, oh, you got to put it in an arc.

... Arc<T>/Rc<T>

Interviewer
Right.

Participant
So that's a lot of where those sort of things come from. In the case of some NIC drivers, right, Ethernet device drivers, you have like a series of cues, descriptor cues for receiving and transmitting buffers.

Interviewer
Right.

Participant
So you often see, you know, you often think, oh, it'd be nice to represent that by a Vec, but they're not on the heap. They're in hardware.

Interviewer
Right.

115
139 **Participant**

And typically we have some of them use virtual addresses, some of them use physical addresses.

Interviewer
Right.

Participant
So sometimes you need to go back and forth between virtual and grab it, grab the address that you passed in previously. You know, once the hardware says, hey, there's been a frame received at this point via DMA.

12:35... Interacting with hardw

Interviewer
Right.

Participant
And, you know, that's sort of another just, I got to trust it type thing.

... Tacit Knowledge

Interviewer
Right.

Participant
So assuming that you set it up and configured it correctly, you know, the address you get back out of that is something that you then reconstitute into a pointer if it's virtual or you figure out which frame, which page was mapped to that frame and grab the virtual address from there. And, you know, it can be complicated to properly handle the transitions to and from the hardware and the software layer when you're talking about going between a safe rust type like a Box and an unsafe type like a pointer.

12:44 So assuming... Box<T> Difficult FFI

Interviewer
Gotcha. Yeah. So I guess in addition to Box, have you had any issues surrounding use of like the unsafe cell drives types?

Participant
Oh, okay. So that would only come up when I'm implementing one of these smart pointers or some kind of container. I have not had any issues. The only case where we use those is when we're implementing synchronization primitives like a mutex.

Interviewer
Right.

Participant
So I think in that case, the usage of the unsafe cell is very clear.

... UnsafeCell<T>

Interviewer

Right.

Participant
You basically spin on an atomic lock and then if you're successful, you know, in the compare exchange, then you can return a reference to that data. And of course, that's unsafe.

12:91...
Atomic Intrinsics
No Other Choice
UnsafeCell<T>

Interviewer
Right.

Participant
I don't believe I have used unsafe cell in any more complicated context than that, but I can do a quick search of our code base. See if I use unsafe cell anywhere.

Interviewer
Okay.

Participant
Ah, okay. Here we go. And thread locals. Yeah, I forgot about that. Oh, okay. And we have two other cases. So being the operating system, we also implement support for thread local storage.

Interviewer
Right.

Participant
So you can... Thread local storage, basically, I don't know if you're familiar with it, but it's a static variable. I'm talking about the ELF native thread local storage. It's a static variable that you annotate with the thread local attribute.

Static Variables

Interviewer
Right.

Participant
And that will allow you to access it. It's like a static that you can access in a single threaded context.

Interviewer
Right.

Participant
So you don't need... Excuse me. You don't need it to be sync.

161

Interviewer
Right.

Participant
In Rust terms.

Interviewer
Right.

Participant
So you can, instead of locking something with mutex, you can lock it with a ref cell.

12:...
Mutex<T>
RefCell<T>
Sync

Interviewer
Right.

Participant
The thread unsafe version of that. Or I should say non thread safe version of that. So in that case, cell and ref cell are frequently used. Cell for types that are copy, ref cell for types that are not. But implementing... If you want to implement something like what the standard library provides, a thread local macro for, which is dynamically lazily initialized thread locals that also have their destructor run, because typically static variables don't have their destructor run.

:
Cell<T>

Interviewer
Right.

Participant
You need to use unsafe cell for that. So you need to... It's basically the way that something like once cell works.

12:...
OnceCell<T>

Interviewer
Right, yeah.

Participant
So I don't need to say anything more about that. It's identical to once cell effectively. And then the other place that we use it is for a lock free queues.

12:...
Unsafe Data Structure

Interviewer
Right.

Participant
Something like what you would see in [library] project.

Interviewer
Right.

Participant
I probably also don't need to get into that. But I mean, you use pointers because again, this is something that you have to use to compare and exchange atomic instructions for, and you can't pass in a high level type. You got to pass in an integer. So that becomes a pointer. Right. And if it's like a linked list or something, an atomic list, it's all pointers. So the unsafe cell is there surrounding the head or the tail pointer, for example, depending on what data structure you're accessing. But again, I think these are pretty well known algorithms. So as long as you're not making horrendous mistakes, the outer interfaces are just like back, you know, new push pops, something like that, or just like that deck. But nothing too crazy. I really strive. Like I said, part of the charter of the project is to not use unsafe. So we're not going to opt for UnsafeCell if we can use like an Rc/Arc and a mutex potentially more cost in 99%. It makes sense. And then were there other container types? Sorry. I was going to say other container types you're interested in.

12:50 But... Atomic Intrinsics
Transmute
12:51 An... UnsafeCell<T>
12:93 But... Arc<T>/Rc<T>
Mutex<T>
UnsafeCell<T>

Interviewer
Oh, I think you pretty much covered all of them. I think the two that you didn't mention were maybe on a net and manually dropped.

Participant
Oh, yeah. Okay. So I don't think we use MaybeUninit but we should. So that is a choice that definitely harms performance. The most common case, again, not unique to this particular domain, but widely known in the Rust community is reads of uninitialized data or sorry, reads of data into an uninitialized array, right? You can't really do that. I know there have been a lot of proposals to make this happen and make it a little safer. But right now you got to do it manually. So in a lot of the cases, we just fill up a buffer with zeros and do it the old fashioned safe way, which I'm sure is slower. Have to do it, though. And the other one you said was manually dropped. Definitely. We do use that. I know that we use that more before whatever version of Rust where they had a guaranteed drop order based on the declaration order of fields in a struct. Before that, we had to use manually drop everywhere that we wanted to impose an ordering on how things were dropped. Now, let me see, where do I use it? So now we use it in some synchronization primitives like read write blocks to make sure that we are dropping the guard before we... So we have some weird read write lock primitives where we're getting additional information about the number of readers and writers for efficiency reasons. So we want to make sure that we drop the guard before we count how many readers there are currently, things like that. I think it's exclusively within synchronization primitives where we're using manually drop now. I know we did use it more often. Oh, yeah. So here's an example. We have a callback for customizable mutex behavior, and we want to make sure that we drop the guard and then invoke this callback function that allows someone to know

12:52 So I don't thi... MaybeUninit<T>
Shifting Ground
12:53 nd the other... ManuallyDrop<T>
12:54... ManuallyDrop<T>
No Other Choice

when a mutex has been dropped, for example, or something like that. So there isn't a way to represent that as far as I know without manually drop, but perhaps I'm missing something. And then we use it in some... I just realized I missed some. We use it in some devices. Let me see what this is. No, that's old. Yeah, no, I think that's it. This is old stuff. That is... Okay, no, here's one. Okay. The only other one. So the Rust standard library has this ability to catch a panic called catch unwind. So us being the operating system, I also need to implement that. So they use ManuallyDrop, and I correspondingly use mainly drop in the same way, because it goes through an intrinsic called the tri-intrinsic, which is basically what will result in the compiler creating an unwind trampoline such that you can actually catch and unwind. And it'll set up all the entries for unwanted to properly work with that. And again, it only uses pointers. So I need to create a struct that has that context, make sure none of them are dropped before I give it to the struct. And then once an unwind is caught, a panic is caught, it calls a callback that I have passed into it. And then I need to make sure that I'm getting the information out of that struct in the right order, and then return it to the user through the safe interface. And you know what? As I'm saying this, maybe we could avoid that now that the order of fields is known, the drop order of fields. But I'm not sure. I would need to get back to you on that. Yeah, we might be able to avoid that. I'm not sure. Oh, we could avoid it. Okay, here's how we could avoid it. So MainlyDrop has this API called take, right? You could wrap it in an option, but then you have a panic possibility, which is technically wrong. So in this case, what we're doing is we're preferring unsafety over the ability of a panic that should never happen. So I guess that's another classification of use cases there. I'm preferring unsafety over a place where I could use an option, but it would be logically invalid for that option to ever not exist. So we don't use an option. 216

12:... ManuallyDrop<T> No Other Choice

12:55 So the Rust... Intrinsics ManuallyDrop<T>

12:56 Oh, we could av... Easier or More Ergono Option<T>

217

218 **Interviewer**
Gotcha. Yeah, that's really interesting. I guess I have a couple sort of final questions. One thing I'm wondering about is with all of these raw pointer conversions, it seems like it could be really useful to have a tool like Miri working in this context where you can track stacked or I guess tree borrows now information about the permissions associated with the raw pointers you're using. But I notice you didn't select that as a bug finding tool that you use. Which I guess makes sense because if you're working in that hardware-dependent context, there are a lot of things Miri wouldn't support for you, probably, right?

219

220 **Participant**
I personally just haven't gotten into it. I'm aware of the pointer provenance work. I don't know if that's related to this. I will say a lot of the places where we descend into raw pointers is usually just obtaining a raw pointer, for example, from say a Box or from some other struct field, right? The pointer to that struct field and then we're passing it into hardware. Rare are the scenarios where we're actually obtaining a raw pointer from somewhere else and then up converting it back into a safe type, right? That does happen and unwinding, but it's rare, especially for hardware. Typically, it's just unidirectional from safe to the raw pointer such that

12:57 I wi... Box<T>

12:94 Rar... Different FFI Memory I

hardware can then use that as a virtual address, right? That's the vast majority of what we're doing. That's just the interface that most devices expose. I don't know if that answered the question, though. You were asking why don't we use Miri or have we experimented with Miri? I have not. I'm well aware of it. I was under the impression that Miri is kind of better for like, you know, I don't know. So, here's a place we could use like implementations of data structures to make sure that you're implementing one of these cute types to make sure that you're not making soundness errors. I probably should and we just have not gotten around to it. Started this project before Miri existed, have never adopted Miri. Like I said, we're doing some of those verification efforts, but that's a separate work that I'm only partially involved in, so I can't speak to it too much. But yeah, we should probably use Miri. I guess I just don't know exactly how to apply it. In an OS, there's umpteen entry points, so it's not clear how I would do. I'd give Miri the address of an interrupt handler and say start from here and give me any soundness errors that could occur from code reachable from that point. If that's how it works, yeah, I could do it. I just haven't bothered, I guess.

12:... Different FFI Memory

12:95 But yeah,... Unaware of Tools

Interviewer
Gotcha. Interesting. Another question here. I guess I was more interested about the provenance stuff, but the way you described that was interesting, how it's mostly unidirectional, where you're exposing things to the purpose of giving it to hardware and not having this sort of round trip, unsafe back to safe conversion.

Participant
Yeah, I mean that happens, but it's rare. But yeah, what were you saying about the point of provenance? You're wondering if we had used that. No, I haven't experimented with it. I'm just aware of the project.

Interviewer
Oh yeah, you answered the question pretty much, like everything I could think of to ask you answered.

Participant
Okay, yeah. I wanted to explore provenance in a little bit more detail, but like I said, we try to maintain safety as much as possible, so I'm not really passing raw pointers around between subsystems or something. So provenance of a pointer is metadata. Yeah, I think it's just wrapped up into the fact that we're using safe types. I don't know. If I think of anything, I can send a follow-up email, but off the top of my head, nothing comes to mind where pointer provenance could... I guess I don't know enough to like qualify myself as someone who could authoritatively speak on whether or not pointer provenance could be beneficial to our project.

Interviewer
Gotcha.

Participant
Maybe. It's admittedly a very complex topic, and it's new to me. And yeah, I've talked to Ralph a little bit. I followed some of his work, but a lot of that shit was just way over my head. But yeah, that's sort of how I feel about that. Yeah, it's tricky.

Interviewer
I guess, are there any problems that you have encountered in your work with Unsafe that you feel like your current development tools can't really adequately solve for you?

Participant
What development tools do we have to help with Unsafe? I'm not really aware of any, aside from, I guess, some formal verification efforts.

Interviewer
Gotcha.

Participant
Man, I don't know how to answer that. That's a tough one. Can you give me some examples of tools? I mean, obviously there's Miri, right? I guess... That can help with Unsafe. The sharp edges around Unsafe?

Interviewer
Yeah, I'd say Miri is the only one that I could say is dedicated particularly to Unsafe. But my guess is that the formal verification efforts you're talking about are likely going to be the answer for the way that you described how that would help you with various different subsystems.

Participant
That makes sense. So admittedly, that's less about verifying whether or not Unsafe code is correct as what the Rust Belt project is doing. That's not what we're doing. It's more about verifying that our implementation of a particular algorithm, like an allocator, is correct. And there's not really Unsafe code in the allocator itself, but the implementation needs to be correct such that when we bridge the Safe allocator code to the Safe higher-level type code, like Box and the alloc types that use the allocator in the middle, which itself is Unsafe, the allocator interface, that we're abiding by that contract and not, for example, duplicately allocating the same memory twice or something that would be invalid. Or, you know, disobeying alignment requests or something like that. Miri, yeah, maybe Miri would be able to help with that. It just occurred to me perhaps another area of tools that could help would be like sanitizers or something. You know, Valve or some other kind of... I know there's a variety of different sanitizers out there. I have not run those. In my view, Rust is generally supposed to relieve me from the obligation of running that stuff, although we do use Unsafe, so it's maybe not true. I have not experimented with those. I guess that would be better for, you know, determining whether or not we accidentally had a leak or something like

12:96 And there's...
Contract or Invariant

12:58...
Tacit Knowledge

that. This is not related to Unsafe, but we have encountered deadlock before, and that is always very difficult to debug, especially with spinlocks, which are required in the early or lowest level parts of the system before we have a task management subsystem. So a tool for that would be helpful, but that's not technically unsafe. It's just really obnoxious.

12:60 This is no...
Wants Solution

Interviewer
Gotcha, gotcha. That makes sense.

Participant
Yeah, if there was a tool that I think was easy to integrate into the project and could... I'm not sure what it would be. We use Clippy, right? There was a Clippy-like tool that said, hey, I notice you're doing something here with this Unsafe. Are you sure that you've actually made sure that all the invariants that you want the Unsafe region to go to to abide by are actually checked? And that's sort of the purpose of formal verification, like you were saying. But yeah, formal verification is a whole issue. It's just really tedious. So I don't know if there was something easier, like on the Clippy integration level difficulty that would for sure be welcome. I could use it. Maybe Miri is that tool. I don't know. Yeah, I feel like Miri is tricky because it has limited support for code that is not Rust. But there is an extension to Valgrind that someone's working on where it will support the same checking as Miri. So maybe that would be useful. Everything we write is in Rust. I mean, there's some inline assembly here and there. But I would just assume that Miri wouldn't check that stuff. I have no idea. Maybe it can be.

12:61 But...
Wants Solution
12:...
Wants Solution
12:...
Inline Assembly

Interviewer
Yeah, I don’t how well Miri supports inline assembly. So yeah.

244

252 **Participant**
But it's minimal.

253

254 **Interviewer**
Gotcha.

255

256 **Participant**
Yeah. It's rarely.

257

258 **Interviewer**
Gotcha. Gotcha. Well, I think that covers like every question I could think of. Really appreciate your time.

259

260 **Participant**
Of course. Yeah, no problem at all. Yeah. If you need follow ups or anything or you need code examples, so everything I have is open

source, you can always take a look, feel free to email me if you have more questions or something.

Interviewer
Yeah, and then I guess I have a couple of follow ups.

Participant
Yeah, no thanks. I was just going to say, it seems pretty cool. What are you, what are you planning to do with this, this like study is it going to motivate some future work or something or TBD.

Interviewer
Yeah, so my area is generally in, I am kind of a verification guy. And I am trying to figure out what the problems are currently surrounding use of unsafe, and sort of like how unsafe is used more broadly, because there have been a lot of studies doing like quantitative analysis of unsafe, how much it's used where it's used but not about like the actual experience of having to work with it. So I'm hoping to get as many perspectives as possible. And I am combining that with some work to I have an extension to Mary where I'm running that on some foreign code bases that makes like see and rust to see if there are bigger issues that happen with like cross language memory management but yeah so I'm kind of hoping this will motivate more work and verification to hopefully help make better verification tools for for unsafe.

Participant
Okay, great. Yeah, we're very interested in that that sort of thing. It'd be nice to lend some formal credence to that and prove that we can actually do so. Yeah, so yeah, I'm very interested to hear the follow ups.

Interviewer
Yeah, no all yeah all communicating with anyone who is involved with the study when we eventually publish something and if you have any questions at all at any point feel free to reach out.

Participant
Yes. Are you involved in the formal methods group at all on that there's a I don't know if you get involved in the rest of the channel there's a formal methods working group that meets any monthly and they have presentations on various formal verification tools that people are working on like any other ones.

Interviewer
I haven't really yet. I probably should be. Yeah, send me a link. No, that'd be awesome. 260

276 **Participant**

I think you should definitely be welcome. Yeah, it's a cool thing I just joined it like last year, and some of the rest folks are working on it like their own. Okay. Well, yeah, great. Anything else. Any other questions.

276