

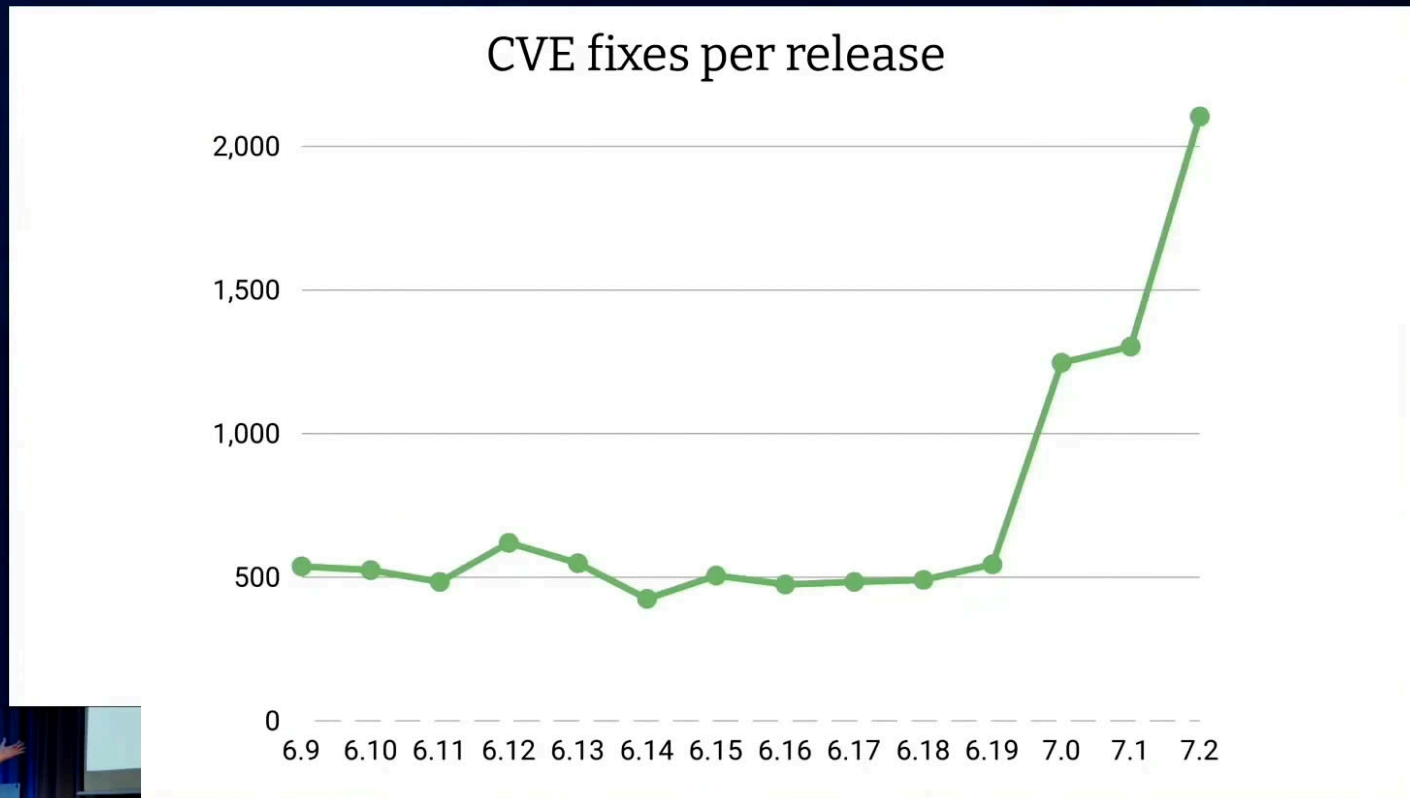
leveraging Field Projections in the Kernel and beyond

Benno Lossin

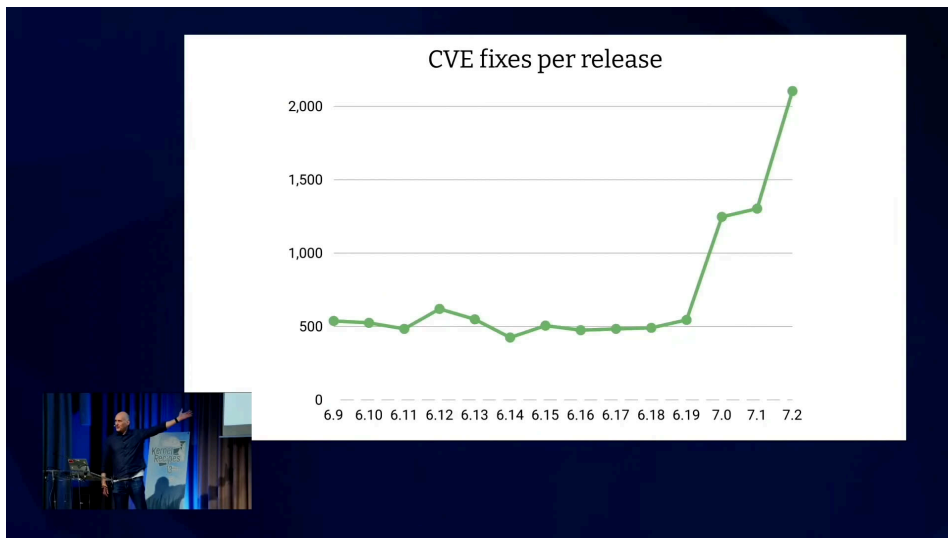
6th october, Prague

but why Rust?

but why Rust?



but why Rust?



- Rust in Android: move fast and fix things — Google, Nov 2025
“Rust 1000x safer than C & C++”
- Rust is now a Tier-1 language for internal development
— Microsoft, Sep 2026

- growing the community

why is Rust safe?

being smart only once, catches many mistakes.
you only need to create
a *good* API once for *every* user of the API to benefit
from the mistakes it prevents.

Rust's superpower

forbid invalid code from compiling

```
fn pointer_to_local(fav: i32) → &'static i32 {  
    let my_value = fav - 42;  
    &my_value  
}
```

```
error[E0515]: cannot return reference to local variable `my_value`  
→ src/main.rs:3:5  
3 |   &my_value  
  |   ^^^^^^^^^ returns a reference to data owned by the current function
```

forbid invalid code: values must always be valid

- all local variables must be initialized before use

```
fn uninitialized_local() → i32 {  
    let x: i32;  
    return x;  
}
```

```
error[E0381]: used binding `x` isn't initialized  
→ src/main.rs:3:12  
|  
2 |     let x: i32;  
|     - binding declared here but left uninitialized  
3 |     return x;  
|           ^ `x` used here but it isn't initialized
```

forbid invalid code: values must always be valid

- when creating a struct, all fields must be initialized

```
struct Driver {  
    debug_level: u8,  
    registration: subsystem::Registration,  
}  
  
fn missing_field() → Driver {  
    return Driver { registration: subsystem::Registration::new() };  
}
```

```
error[E0063]: missing field `debug_level` in initializer of `Driver`  
  → src/main.rs:6:12  
  |  
6 |     return Driver { registration: subsystem::Registration::new() };  
  |                   ^^^^^^^ missing `debug_level`
```

forbid invalid code: ownership and automatic cleanup

```
fn create_driver() → Box<Driver> {  
    let mut driver = Box::new(Driver::new());  
    //          ^^^^^^^^ allocates memory  
  
    driver.config.len = 42;  
    // ...  
  
    return driver; // pass ownership of the allocation to the caller  
}
```

```
fn use_twice() {  
    let driver = create_driver();  
    consume_driver(driver);  
    consume_driver(driver);  
}
```

forbid invalid code: ownership and automatic cleanup

```
fn use_twice() {  
    let driver = create_driver();  
    consume_driver(driver);  
    consume_driver(driver);  
}
```

error[E0382]: use of moved value: `driver`

→ src/main.rs:10:20

```
|  
8 |     let driver = create_driver();  
|     ----- move occurs because `driver` has type `Box<Driver>`,  
|           which does not implement the `Copy` trait  
9 |     consume_driver(driver);  
|           ----- value moved here  
10 |     consume_driver(driver);  
|                       ^^^^^^^ value used here after move  
|
```

Rust's even better superpower: custom *restrictive* APIs

- prevent calling functions in certain situations:
 - if it would be invalid according to the current driver model state
 - only give access to data behind a lock if it's locked
- make invalid state unrepresentable:
 - bounded integers to prevent passing out of range values
 - only provide access to data if it exists

again: code that violates these properties **does not compile**

being smart once, catches many mistakes.

why can't we just make C safer?

it's missing too many vital and intertwined features:

- mark code as safe
- field privacy
- enums (safe unions)
- initialization tracking
- ownership tracking
- hygienic and AST-based macros
- lifetime tracking
- lifetime generics
- type generics
- trait equivalent
- a library using these
- ...

this is Rust.

we should continue improving C: it will make our existing code safe, enable smoother interop with Rust, and help C in general.

Field Projections

many pointer types in Rust

- temporarily loaned: `&'a T` and `&'a mut T`
- allocation-owning: `Box<T>`
- raw pointer: `*mut T`
- raw with optimization: `NonNull<T>`
- atomic reference counted (owning pointer): `Arc<T>`
- intrusive reference counted: `ARef<T>`
- owning pointer to DMA memory: `Coherent<T>`

smart pointer support

```
pub trait Deref {  
    type Target;  
  
    fn deref(&self) → &Self::Target;  
}
```

```
impl<T> Deref for KBox<T> {  
    type Target = T;  
  
    fn deref(&self) → &Self::Target {  
        unsafe { &*self.ptr }  
    }  
}
```

smart pointer support

```
pub trait DerefMut: Deref {  
    fn deref_mut(&mut self) → &mut Self::Target;  
}
```

```
impl<T> DerefMut for KBox<T> {  
    fn deref_mut(&mut self) → &mut Self::Target {  
        unsafe { &mut *self.ptr }  
    }  
}
```

limitations of Deref[Mut]

- cannot be used for pointers with fewer guarantees than &T, like:
 - NonNull<T>
 - Coherent<T>
 - Mmio<T>
 - ValidPtr<T>
- the borrow-checker doesn't understand Deref natively

places in Rust

- a *place* is a location in memory
- *place expressions* allow referring to a place via syntax:
 - local and static variables: `x` and `GLOBAL`
 - dereference: `*place`
 - field access: `place.field`
 - indexing: `place[idx]`
- *place operations* allow modifying/using a place:
 - read: `let value = place;`
 - write: `place = 42;`
 - borrow: `&place` and `&mut place`

Field Projections = custom place operations

- Rust has traits to customize the operators:
 - for example, the + operator:

```
pub trait Add<Rhs> {  
    type Output;  
  
    fn add(self, rhs: Rhs) → Self::Output;  
}
```

- Field Projections is about
 - adding traits to customize place operations and expressions
 - and make the borrow checker understand them
 - and ensure that they interact nicely with all other features of Rust

design internals

- no user has to know these or interact with them,
- rather complex, so not covered here

design internals

Benno Lossin & Nadri

He/Him & He/Him

Field Projections
— Making
Custom Pointers
feel Builtin



who is working with me on this?

- Tyler — language team member
- Xiang — compiler team member
- Nadrieril — language team advisor
- being supported by several Rust maintainers
- and many Rust contributors on Zulip

Kernel use-cases

“standard” pointers

- Coherent<T>
- Mmio<T>
- &Untrusted<T>
- NonNull<T>
- ValidPtr<T>
- KBox<T>, UniqueArc<T>, ARef<T>, ... many more

```
fn project_field(  
    ptr: ValidPtr<bindings::bio_vec>,  
) → (ValidPtr<*mut bindings::page>, ValidPtr<ffi::c_uint>) {  
    (@ptr.bv_page, @ptr.bv_len)  
}
```

exotic use-cases: safe, zero-cost RCU abstractions

- already can use simple RCU in Rust:

```
let rcu_protected_data: &Rcu<Box<Data>> = ... ;  
let guard = rcu::read_lock();  
let data: &Data = rcu_protected_data.read(&guard);
```

- but: more complicated use-cases are unsafe:
 - often have a lock, like a `Mutex<T>` to synchronize writers,
 - but readers want to bypass the mutex
 - Rust forbids access to the value without locking the mutex
 - field projections allows safely bypassing the mutex

exotic use-cases: custom layout

- bitfields

```
#[bitfields]
pub struct Flags {
    active: u1,
    blocked: u1,
    reserved: u6,
}
```

```
fn demo(flags: &mut Flags) {
    flags.active = cv!(1);
    flags.blocked = cv!(0);
}
```

exotic use-cases: custom layout

- multiple dynamically sized array fields

```
#[dyn_array_fields]
pub struct NegotiateAlgsRsp {
    version: u8,
    ext_asym_sel_count: u8,
    ext_hash_sel_count: u8,
    ext_asym: [u8; self.ext_asym_sel_count],
    ext_hash: [u8; self.ext_hash_sel_count],
}
```

exotic use-cases: virtual places

- access hardware registers instead of fields

```
#[registers]
pub struct Registers {
    #[reg(read = my_dev_read_status, write = my_dev_set_status)]
    status: u8,
    #[reg(write = my_dev_reset)]
    reset: bool,
}
```

- track the access path of a user in the type system

thanks for attending!
