C++ Interop with Rust

# Agenda

- Introduction
- Exercises (manual, bindgen, cxx, autocxx, ipc)
- Reflection and Comparison
- Case studies (Android, Chrome, fish-shell, etc.)

# Motivation

- Many reasons to use Rust (e.g. safety, performance, concurrency, ...)
- You can't rewrite your whole codebase
- Integrate Rust into C++ projects (and vice versa)
    - **Code**: Calling code from other language
    - **Build system**: Everything works together

## C Interop

- C is the lingua franca of programming languages
- Rust doesn't have a stable ABI, neither does C++
- Solutions like `bindgen` and `cbindgen` help to generate bindings

C++ Interop

- Exceptions
- Templates
- Complex types (String, Vec, …)
- Memory management
- Inheritance
- No stable ABI
- Build system integration
- …

# Libraries

Overview

- Rust -> Cpp
  - bindgen
  - cpp
- Cpp -> Rust
  - cbindgen
  - BuFFI
- Cpp <-> Rust
  - cxx
  - autocxx

# Others

- crubit: C++/Rust interop library by Google
  - Experimental, expect breaking changes
- diplomat: Supports C, C++ and JavaScript.
  - Unidirectional, when foreign code wants to call Rust libraries
  - Experimental
- uniffi: Supports Kotlin, Swift, Python, Ruby, Go, C#
  - Uses a IDL instead of macros
  - They are open for new adapters
- cglue: Bridges Rust traits between C and other languages.

# Exercises Overview

- Repository can be found here:
  - https://github.com/not-matthias/cpp-interop
- Contains the skeleton, solutions and slides

Let's get started!

Rust vs C++

What is a "move":

- **Rust**: memcpy
  - (similar to the C++ `std::is_trivially_moveable` type-trait).
  - Moves also render the moved-from object inaccessible.
- **C++**: Like a `Clone` operation
  - Leaves the moved-from value accessible to be destroyed at the end of the scope.

Rust vs C++

This results in an important difference:

- **Rust**: Compiler can move data around (memcpy). The object doesn't know.
- **C++**: Object stays where it is, until it has its "move constructor" invoked.

Problem: C++ objects can have self-referential pointers

- Very common (even in some impls of `std::string`)
- Rust doing a memcpy would invalidate such a pointer

# Rust vs C++

```cpp
struct A
{
    int a;
    int b;
    int* p{&a};
};

int main()
{
   auto p = std::make_unique<A>();
   A a = std::move(*p.get());  // gets moved here, a.p is dangling.
}
```

# Where can objects be created?

- C++ Heap (using `std::unique_ptr`)
- Rust Heap (using `Box`)
- Rust Stack (using `moveit!`)

# Case Studies

Chromium

> We are pleased to announce that moving forward, the Chromium project is going to support the use of third-party Rust libraries from C++ in Chromium.

- Supporting the Use of Rust in the Chromium Project, 2023

Goals:

- Simplify the code
- Speed up development: Less code to write
- Improve security (decreasing bug density)

## How do they want to do it?

- Interop only in one direction: from C++ to Rust
  - Majority of stack frames are in C++
  - Rust can not land in arbitrary C++ code, only in functions passed through the API from C++.
- Only support for third-party libraries
  - Standalone components: No implicit knowledge about Chromium
  - Simple APIs, focused on a single task
- Google invests into Crubit to explore interop
  - They don't want to write a `#[cxx::bridge]`. Everything should be automatic.

# Android

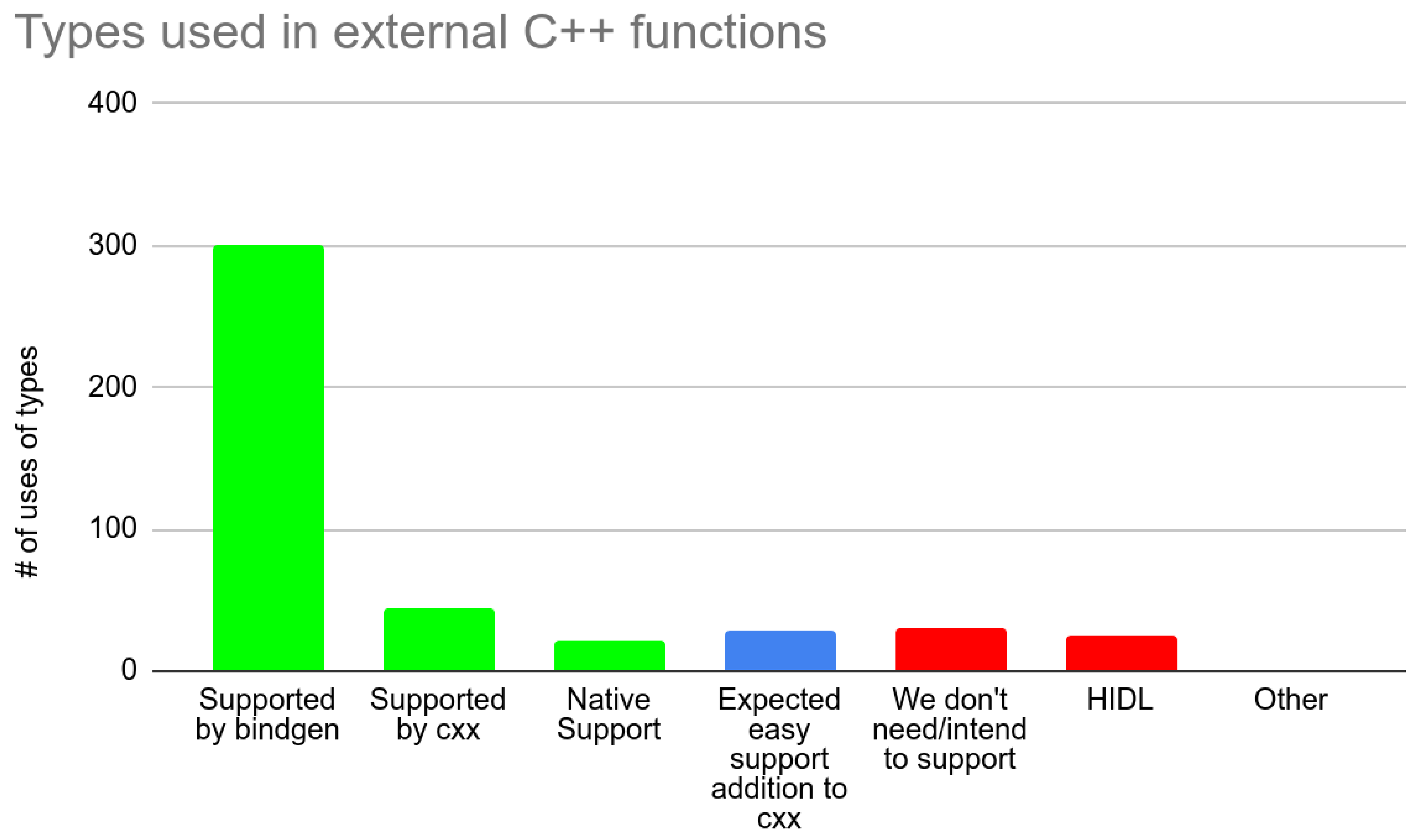
> [...] we have too much C++ to consider ignoring it, rewriting all of it is infeasible, and rewriting older code would likely be counterproductive as the bugs in that code have largely been fixed. This means interoperability is the most practical way forward.

- Rust/C++ interop in the Android Platform, 2021

Goals:

- Rust must be able to call functions from C++ libraries and vice versa.
- FFI should require a minimum of boilerplate.
- FFI should not require deep expertise.

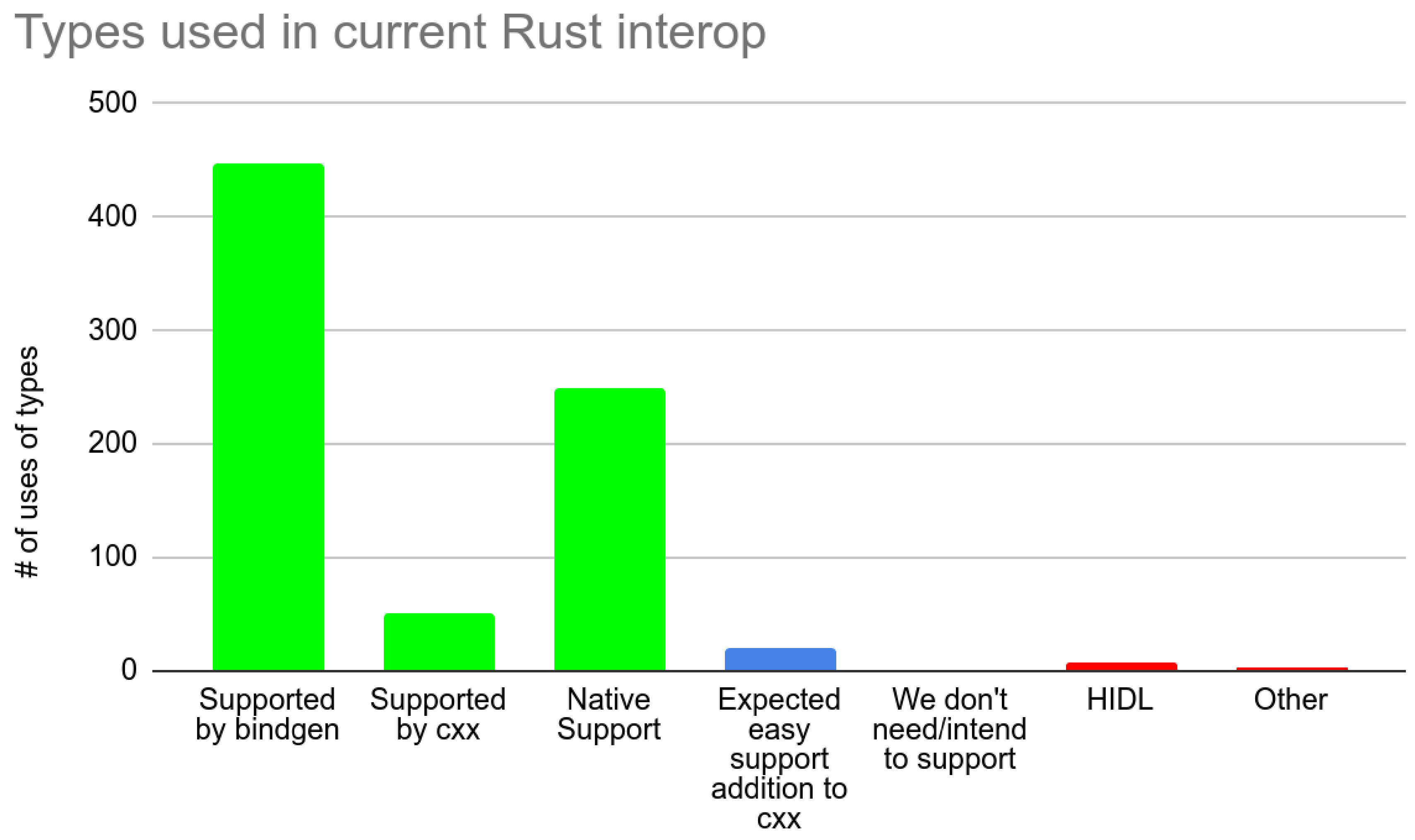
## Analysis of most commonly used modules



## Analysis of most commonly used modules

- **Analyzed Modules**: liblog, libbase, libutils, libcutils, libhidlbase, libbinder, libhardware, libz, libcrypto, and libui.
- Over 81% of types are fully supported (first 3 categories)
- For 7% of types, support can be easily added (4th category)

## Analysis of Rust/C++ Interop in AOSP



## Analysis of Rust/C++ Interop in AOSP

- `bindgen` handles the majority of the types
- Native support with AIDL and protobufs are simple and ergonomic to use
  - Often difficult to use tools like `cxx` or `bindgen`
- Auto-generated interop solutions are not always an option:
  - Manually creating C-wrappers is simple and straightforward
  - Wrappers are then passed to `bindgen`
  - Used in profcollectd

Fish-shell

- "the friendly interactive shell"
- Decided to rewrite it in Rust
- They considered many different languages like Java, Zig or Go
- Incremental rewrite in Rust

## Why did they want to switch?

- "Nobody really likes C++ or CMake"
  - No clear path for getting off old toolchains.
  - Every year the pain will get worse.
  - (Thread) safety issues
  - Lack of tooling
- **Why switch to Rust?**
  - Active and growing community
  - Needed for concurrent function execution
  - Will help to be perceived as modern and relevant

## How did they do it?

- `autocxx` (for using C++ types in Rust)
- `cxx` (for basic C++ <-> Rust interop)
- `corrosion` for CMake integration

## How did they do it?

- Used `autocxx` to port one component at a time
- Add FFI glue to C++, to make it callable from Rust (where necessary)
- Call rewritten components from C++
- Had to wrap some C++ variables in `std::unique_ptr`
    - To make the ownership rules understandable to `autocxx`
    - Caused a small perf drop

# Summary

Library	Rust -> C/C++	C/C++ -> Rust	Automatic
bindgen	✅	❌	✅
cbindgen	❌	✅	✅
cxx	✅	✅	❌
autocxx	✅	✅	✅
ipc	✅	✅	✅

## When to use which library?

- **bindgen**: If you are making bindings to C code, or basic C++
- **cbindgen**: If you want to generate C/C++ headers from Rust code
- **cxx**: If you want a bidirectional bridge between C++ and Rust, with support for C++/Rust types.
  - When unrestricted changes to C++ code are possible
- **autocxx**: `cxx`, but with automatic bridge generation
- **IPC**: If you can use IPC, use it. No need to deal with FFI.

# When to use which library?

1. Use IPC where possible
2. Try autocxx
3. Try cxx
4. Write custom C glue code

## That's it!

Repo: github.com/not-matthias/cpp-interop

Contact: matthias.heiden@protonmail.com