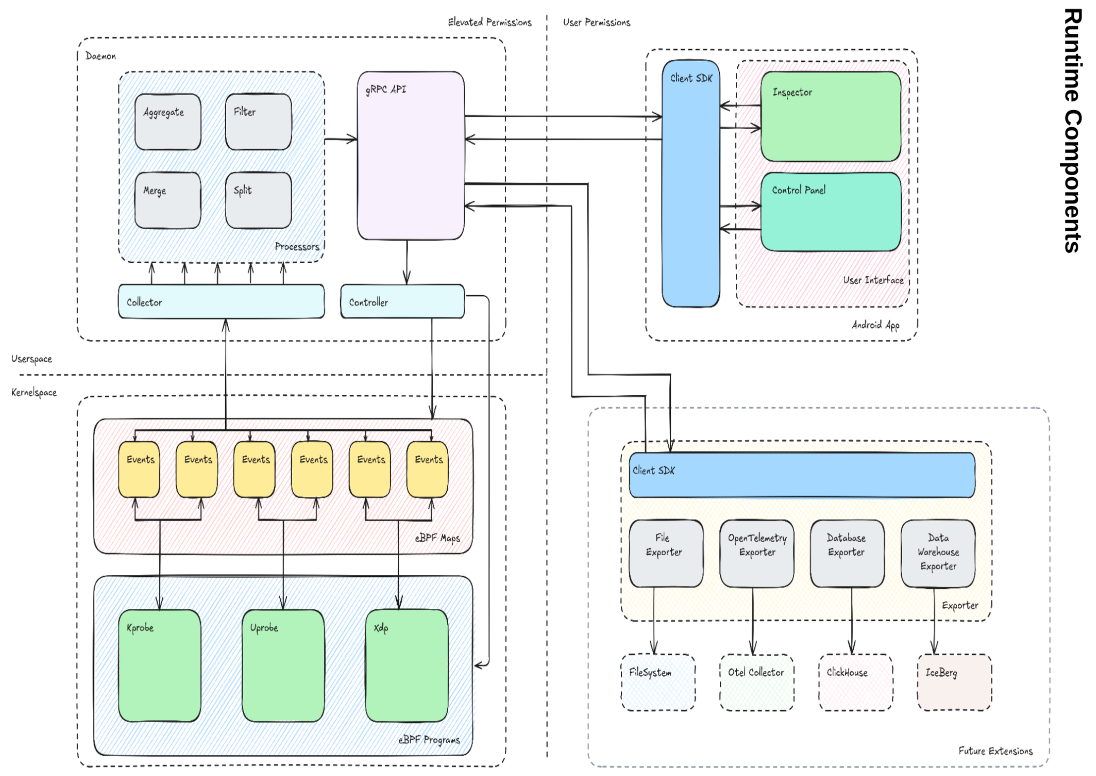
Runtime Components



Components

Our Architecture is made up of four smaller components to allow for possible extension in the future.

eBPF Programs

A single rust crate contains all the eBPF functionality, this is called `backend-ebpf`. It defines multiple programs for collecting different kinds of events:

- [x] calls to `vfs_write`
- [x] data sent over unix domain sockets
- [ ] arbitrary userspace functions
- [ ] arbitrary kernelspace functions
- [ ] usage of `futex`

- [ ] and more see [goals document]

These eBPF programs are configurable at runtime, allowing dynamic attaching and detaching of programs. This makes it possible to adjust what events and from what apps to collect.

For communication between Userspace and eBPF programs, maps are used. The main variants are the `BPF_MAP_TYPE_HASH` and `BPF_MAP_TYPE_RINGBUF` .

The first one is used for configuration of our eBPF programs. As key a combination of `pid` and `tid` is used and the value is the configuration for the specific eBPF program.

The latter is used for sending events from eBPF to the daemon. Strong ordering and `epoll` makes it the ideal map type for collecting event data.

Userspace Daemon

The userspace daemon has four jobs.

Configuration of eBPF programs

Through aforementioned maps, the userspace daemon can configure the eBPF programs at runtime. This capability is exposed through a `grpc` API, to make configuration possible from other applications.

Collecting events

Also through maps, the userspace daemon collects events and sends them to client over a `grpc` API.

Processing events

Between collecting events from eBPF and sending them to clients, events can be further processed:

- [ ] Aggregate
- [ ] Filter
- [ ] Combine
- [ ] For more see [goals document](https://github.com/amosproj/amos2024ws03-android-zero-instrumentation/blob/main/Documentation/

Client SDK

The client SDK offers convenient APIs targeted at developers over the `grpc` infrastructure. It is used by our Android application but can also be used in the future to write other clients.

Possible usecases:

Possible usecases:

- [x] Visualizing events in an Android App
- [ ] Exporting events in Open Telemetry format
- [ ] Sending events to a data warehouse
- [ ] Sending events to databases like Clickhouse
- [ ] Persisting events to disk for later analysis

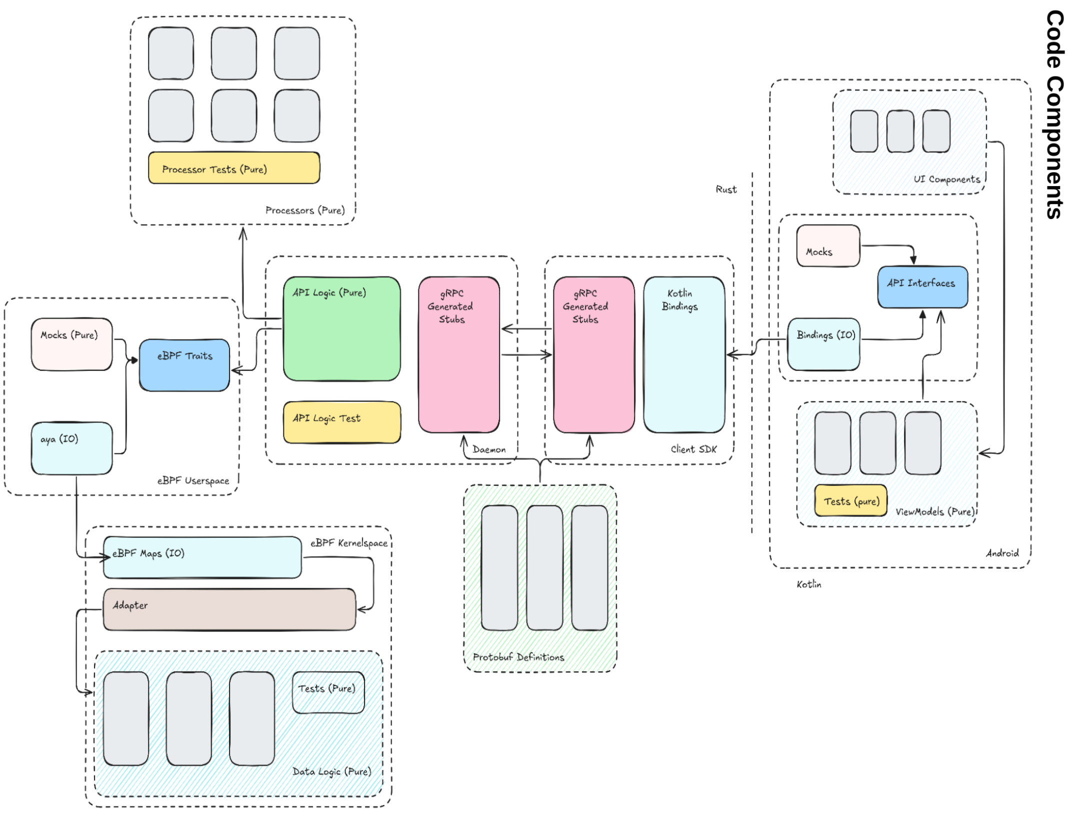
Android App

The android application is used for configuration our eBPF programs using the Client SDK.

It should be possible to:

- [x] Visualization of events
- [ ] Raw event logging

Code Components



Isolation

Isolation has high priority:

- Assigning individual components to individual team members
- Faster build and test cycle
- Easier test design

Abstracting IO

For creating useful and efficient tests, we need to abstract IO operations as much as possible.

Using Rust's traits and Kotlin's interface we can decouple our implementation as much as possible from the real system. This makes deterministic and fast testing possible.

Type safety

Both Rust and Kotlin are type safe languages. For our API we use `grpc` to generate type safe code definitions for both client and server.