ChatGPT

Ensuring Rust eBPF Code Compiles to Valid eBPF Bytecode

1. Emitting eBPF Bytecode vs. LLVM IR Output

When targeting the `bpfel-unknown-none` architecture, it's crucial that the build produces a proper eBPF object (ELF file with eBPF bytecode sections) rather than leaving the output as LLVM bitcode or IR. By default, Rust's BPF targets have historically emitted LLVM bitcode if a proper linker isn't invoked. In fact, a recent Rust issue shows that using `cargo build --target bpfel-unknown-none -Z build-std` without a BPF linker can result in an `.rlib` containing LLVM bitcode instead of machine code. To ensure you get **actual eBPF bytecode** (in an ELF64 with eBPF sections), you must incorporate a linking step that translates IR into eBPF instructions. In practice, this means using a BPF-specific linker (discussed below) or instructing Rust to produce an object file.

One straightforward way to verify you have real eBPF output is to inspect the file. After building your eBPF crate, run the Unix `file` command and an objdump on the artifact. For example, a correct eBPF build yields output like:

- `file target/bpfel-unknown-none/debug/myapp` -> *"ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), not stripped"*
- `llvm-objdump -S target/bpfel-unknown-none/debug/myapp` -> shows disassembly of BPF instructions (e.g. bytecode opcodes like `b7 05 00 00 aa 00 00 00 r5 = 170` etc.).

In an Aya-based project, a successful build of the eBPF program should produce such an ELF object file containing eBPF machine code (typically in sections named after the program, like `.text` or custom sections such as `kprobe/…`, `xdp/…` depending on the macro). Ensuring this outcome often requires proper configuration of the linker and build flags, as described next.

2. Choosing the Right Linker (bpf-linker vs. ld.lld vs. clang)

Using a BPF-aware linker is essential – the recommended choice in the Rust/Aya ecosystem is the [`bpf-linker` tool][74]. The `bpf-linker` is a purpose-built static linker for eBPF programs. It operates on LLVM bitcode and produces a final eBPF object, handling several eBPF-specific needs: it links multiple object/bitcode files, performs link-time optimizations, and applies workarounds for older kernel limitations. For example, BPF bytecode used to disallow function calls and loops on older kernels, so `bpf-linker` can **inline or unroll** functions and loops as needed (with flags like `--ignore-inline-never` and `--unroll-loops`). It also provides implementations or expansions for memory intrinsics (`memcpy`, etc.) when LLVM doesn't automatically do so, ensuring the final code has no unresolved built-ins. These capabilities make `bpf-linker` **the preferred linker** for Rust eBPF projects using Aya.

By contrast, `ld.lld` **(LLVM's linker)** has only recently gained BPF support and isn't yet as integrated or convenient for Rust eBPF workflows. At the time of writing, the Rust nightly toolchain's BPF target does not automatically invoke `lld` for BPF; without `bpf-linker`, it will emit bitcode and fail to produce a usable eBPF binary. Using `ld.lld` manually would require careful invocation and possibly a custom linker script to preserve eBPF sections (e.g., license, map definitions) and to set the ELF e_machine to BPF. In practice, Rust developers avoid this complexity by relying on `bpf-linker`. Similarly, invoking `clang` is not needed in a pure-Rust project – clang is typically used for C eBPF programs, but for Rust, `bpf-linker` (or Rust's built-in integration with it) is the solution to get from Rust IR to final eBPF bytecode. In summary, **use** `bpf-linker` unless you have a very specific reason to attempt `ld.lld`. The Aya maintainers created `bpf-linker` specifically because no existing linker properly handled the Rust-to-BPF workflow at the time. It's recommended in Aya's documentation and by real-world projects as the standard linker for Rust eBPF builds.

Why `bpf-linker`? Beyond producing a correct raw eBPF ELF, it ensures that only supported Rust code ends up in the binary. It links at the LLVM bitcode stage, meaning unused code from the Rust core/alloc libraries can be eliminated before machine-code generation. This avoids errors where Rust might otherwise try to emit unsupported constructs (e.g. unsupported floating point or 64-bit division in older eBPF) that are never actually needed by your program. The linker's design pushes codegen to the last step, after dead-code-elimination and LTO, yielding a minimal and compliant eBPF bytecode. It also accepts multiple input modules (e.g., if your eBPF program is split into library crates), whereas the traditional C workflow would compile everything together via `#include` to avoid linking entirely [1]. In Rust, because you naturally have separate crates, a linker is necessary to combine them [2] – `bpf-linker` fills this role, whereas plain `lld` historically could not link BPF objects at all.

Note: If `bpf-linker` is installed (via `cargo install bpf-linker`), you can tell Cargo/Rust to use it. Typically this is done through Cargo config or flags (see next section), since Rust doesn't yet automatically invoke it on stable toolchains. In the future, as Rust's BPF target matures, we may see built-in support where rustc calls `bpf-linker` or an equivalent by default. Until then, explicitly using `bpf-linker` is the safe choice to get a *"raw ELF with eBPF sections"* as output instead of LLVM IR.

3. Build Flags and Configuration for eBPF Targets

Building eBPF with Rust often requires a few **nightly-only flags and config tweaks** to produce a valid result. Key considerations include the following:

- **Use `#![no_std]` and no main:** Your eBPF crate (such as `lock-ebpf`) should be `no_std` (no OS abstractions) and typically `#![no_main]` (since eBPF programs don't use `main`). This is already done in the user's setup, which is correct for kernel eBPF programs.
- **Explicitly build with core (and alloc if needed):** Because you're not linking the full std, you must include the core library (and optionally alloc). The build is invoked with `-Z build-std=core,alloc` which causes the Rust core and alloc crates to be built for the BPF target. This is essential for even basic functionality, since `core` provides fundamental types and language features. The `alloc` crate is included because the user specified it, presumably to allow use of collections or heap allocation. Keep in mind that actual heap allocation in eBPF is restricted – if `alloc` is used, it likely relies on a limited allocator or is used only in test contexts. Ensure that an appropriate global allocator is provided or that you only use data structures that allocate via BPF

maps or per-cpu arrays. In Aya's case, many examples include `alloc` to support data structures, but direct allocations in eBPF are generally avoided or done through special means. The important part is that you *include these crates in the build*, otherwise even `core::fmt` for printing, etc., wouldn't be available.

- **Set panic strategy to abort:** Unwinding is not supported in eBPF programs. You should compile with `-C panic=abort` so that any panic in eBPF code will just abort (which in BPF typically means a helper call failure or map update failure will be propagated as an error code). In the build output of an Aya template, for example, we see `-C panic=abort` being passed to rustc. If you forget this, the compiler might include unwinding symbols which would make the eBPF program invalid (or simply fail to compile, as unwinding requires libunwind which doesn't exist for BPF). Setting abort ensures any `panic!` just terminates the eBPF program (which usually results in the BPF verifier rejecting the code path, so it's best to avoid panics entirely).
- **Enable LTO and single codegen unit:** It's recommended to use Link-Time Optimization (`-C lto`) and set `codegen-units=1` for eBPF builds. This causes the compiler to optimize across crate boundaries and produce one monolithic module, which the BPF linker can then optimize further. In practice, Aya's build scripts do exactly this – the rustc invocation for the eBPF crate shows `-C lto -C codegen-units=1 -C opt-level=3` in release builds. LTO helps ensure that any inlining or constant propagation that can happen does happen before final codegen, often resulting in fewer instructions (important for the BPF verifier's instruction limit).
- **Specify the linker and linker flavor (if needed):** To ensure Rust uses `bpf-linker`, you might need to set `RUSTFLAGS` or Cargo configuration to use it as the linker. For example, one approach (if not using the built-in target support) is: `-C linker-flavor=wasm-ld -C linker=bpf-linker -C linker-plugin-lto`. This incantation tells rustc to treat `bpf-linker` like a `wasm-ld` compatible linker and to enable LTO at the linker stage. In many Aya projects, however, this is handled behind the scenes. If you have the nightly toolchain with BPF target support, you can instead configure Cargo to use `bpf-linker` for that target (see next section) so that simply running `cargo +nightly build --target bpfel-unknown-none` automatically calls the linker. The key is that **somewhere in your build configuration you must indicate to use `bpf-linker`**. Otherwise, as noted, you may end up with only LLVM bitcode. You can pass this via environment: for instance, `RUSTFLAGS="-C linker=bpf-linker"` (and optionally the linker flavor if needed) when invoking cargo, or via `.cargo/config.toml`.

- **Linker script usage:** In general, you do *not* need a custom linker script for eBPF when using `bpf-linker` or the Rust target defaults. The `bpf-linker` knows how to handle BPF sections and will preserve sections like `.maps` or `.bss` and so on appropriately. If you were to attempt using `ld.lld` directly, you might need to ensure sections aren't stripped, but with `bpf-linker` this is handled. It's more important to ensure that your code uses the correct section attributes via the Aya macros (for example, the Aya `#[kprobe]` macro will place the function in a section named `kprobe/your_function_name` internally). Those section names must be present in the ELF so the loader knows how to attach the programs. Thus, focus on using the macros and let the tooling handle the sections. No manual linker script is typically required in Aya projects. (One special section to note is the license: Aya's `aya-bpf` crate will automatically include the license string section for you – by default Aya programs are licensed GPL or MIT depending on template, ensuring the kernel will load them. Just be sure not to remove or rename that section.)

- **Target feature flags:** The Rust BPF targets generally set appropriate LLVM target features (like `+alu32` for 32-bit ALU mode on older kernels). You normally don't need to manually set `-C target-feature` unless you have a specific reason. If targeting very old kernels (pre-5.3), you might choose to enable `alu32` (32-bit ALU ops) to force 32-bit subregister operations for compatibility, but modern BPF (v2 ISA and above) supports 64-bit ALU. The `bpf-linker` by default uses a "generic" CPU target which should be compatible with all kernel versions (unless you opt into a specific BPF ISA version). So usually no extra `-C target-cpu` or features are needed beyond what the target triple provides.

In summary, the invocation you provided – `cargo +nightly build --target bpfel-unknown-none -Z build-std=core,alloc` – is on the right track. Just ensure that behind the scenes Cargo knows to use `bpf-linker` as the linker (either via `~/.cargo/config` or project config or via an xtask script). Also consider adding release optimizations (e.g., `--release` or setting opt-level) and the panic=abort/LTO flags as above. The Aya templates typically automate all these: for example, their xtask passes `+nightly` and the target and `-Z build-std=core` and sets release mode when needed, and the `.cargo/config` (or RUSTFLAGS) sets `panic=abort` and other rustc flags.

4. Role of `.cargo/config.toml` in Xtask Builds and Target Overrides

The `.cargo/config.toml` file is a convenient place to put target-specific build settings so you don't have to specify them on each command. In eBPF projects, you might use it to define the BPF target, linker, and rustflags. For example, the **bpf-linker README** suggests setting in `.cargo/config.toml`:

```
[build]
target = "bpfel-unknown-none"
unstable.build-std = ["core"]    # add "alloc" if needed
rustflags = ["-C debuginfo=2", "-C link-arg=--btf"]
# (optional BTF flags, only for BPF crate)
```

This would default your builds to the BPF target and include core, etc., so you could just run `cargo build` in the eBPF crate directory. However, **caution**: If your project is a workspace with both user-space and eBPF programs (as Aya projects are), setting a global `[build].target = "bpfel-unknown-none"` in the workspace config will make *every* crate compile for eBPF, which you **don't** want. You only want the eBPF sub-crate to use the BPF target, while your userspace program uses the native target. There are a couple of ways to manage this:

- **Per-crate .cargo/config:** You can put a separate `.cargo/config.toml` inside the eBPF crate's directory (e.g., `lock-ebpf/.cargo/config.toml`). Cargo will use the nearest config for that crate when building it. In the Aya template's earlier versions, they did exactly this: the eBPF crate had a config specifying the target and build-std for itself. This ensures when you build the eBPF crate (either directly or via an xtask), those settings are applied, without affecting the whole workspace.

- **Xtask overrides:** In your case, you have a custom **xtask** to build the `lock-ebpf` crate. The xtask (which is a separate small Rust program) likely invokes Cargo with the appropriate flags. For instance, it might run something akin to `cargo +nightly build -p lock-ebpf --target bpfel-unknown-none -Z build-std=core,alloc`. If the xtask already specifies the target, it may override or obviate the need for `.cargo/config` settings. In fact, the Aya template maintainers removed the hardcoded target from the config once they introduced the xtask, to avoid conflicts. A commit in the Aya template shows they deleted the `[build] target = "bpfel-unknown-none"` from the eBPF crate's config when adding the xtask, relying on the xtask to supply it instead. They kept only a `target-dir = "../target"` to have a unified target directory, but no longer forced the target via config.

- **Linker specification in config:** You can also use `.cargo/config.toml` to specify the linker for the BPF target. For example:

```
[target.bpfel-unknown-none]
linker = "bpf-linker"
```

This would tell Cargo/rustc to invoke `bpf-linker` when linking that target. (If you installed `bpf-linker` with Cargo, it should be in your PATH.) You might still need `linker-flavor = "wasm-ld"` under `[target.bpfel-unknown-none]` if using older rustc trickery, but in recent Rust nightlies, simply setting the custom linker may work since the target is known to be "like an ld". Check the latest Rust docs for the BPF target – if it doesn't recognize `bpf-linker`, using the wasm-ld flavor as shown above is a workaround.

- **Rustflags in config:** The config can also carry `rustflags` for the BPF target. For instance, if you want to always compile the BPF program with debug info and BTF enabled, you could put:

```
[target.bpfel-unknown-none]
rustflags = ["-C debuginfo=2", "-C link-arg=--btf", "-C panic=abort", "-C
lto=yes", "-C codegen-units=1"]
```

This ensures whenever the BPF target is built, those rustc flags are applied. In the Aya template's config (historically) we saw similar flags being set. Just make sure these flags are **only** applied to the BPF target (hence under the `[target.bpfel-unknown-none]` table) so that your host program isn't built with e.g. `panic=abort` unless you intend that.

In summary, `.cargo/config.toml` is a powerful way to simplify building the eBPF code. In an xtask-based setup, the xtask can either rely on the config or override it. The interplay is: **the xtask's Cargo invocation will pick up config settings unless it provides its own**. Command-line flags usually take precedence over config. For example, if `.cargo/config` in the eBPF crate sets the target to bpfel-unknown-none, you could simply do `cargo +nightly build -p lock-ebpf` in the workspace and it would target BPF. But if your xtask already runs with `--target bpfel-unknown-none`, it doesn't hurt to also have it in config (it would be redundant but not conflicting). The potential for conflict is if you accidentally apply BPF target config to the whole workspace. So the **best practice** is either: use a *local config in the eBPF crate* or have the xtask pass all needed flags explicitly and keep the config minimal. Many

Aya projects use a hybrid: the xtask passes `-Z build-std` and `--target`, while `.cargo/config` in the eBPF crate might set some default rustflags (like BTF or panic=abort) so that those are always used. This can make the xtask code simpler.

Finally, `.cargo/config.toml` can define **aliases** which is handy for xtasks. The Aya template, for instance, defined an alias so that you can run `cargo xtask build-ebpf` instead of the longer `cargo run --package xtask -- build-ebpf` command. This alias lives in the config and improves developer ergonomics. It doesn't directly affect the build output, but it's good to be aware of such use.

5. BTF Support in Aya and Its Influence on Output Format

BPF Type Format (BTF) is a metadata format that encodes debug information (essentially a lightweight form of DWARF specialized for BPF). Including BTF in your eBPF binary is highly beneficial for modern eBPF development: it enables **CO-RE (Compile Once – Run Everywhere)** features, meaning your program can adapt to differences in kernel data structure layouts and be loaded on multiple kernel versions without recompiling. Aya fully embraces BTF for this purpose. The Aya documentation highlights that BTF support is "transparently enabled when supported by the target kernel" and allows eBPF programs compiled against one kernel to run on others [3]. In practice, this means if your kernel has BTF info available (most 5.2+ kernels with CONFIG_DEBUG_INFO_BTF do), Aya will leverage it.

To include BTF info in your **compiled eBPF bytecode**, you need to compile with the right flags. This typically means enabling full debug info (`-C debuginfo=2`) and passing a special linker flag to emit BTF. For `bpf-linker`, the flag is `--btf`. The bpf-linker will then generate a `.BTF` section in the ELF containing type info for your program types. You can set this via `RUSTFLAGS` or config. For example:

```
RUSTFLAGS="-C debuginfo=2 -C link-arg=--btf" cargo +nightly build --
target=bpfel-unknown-none -Z build-std=core,alloc --release
```

Or in `.cargo/config.toml` under the BPF target, include `rustflags = "-C debuginfo=2 -C link-arg=--btf"`. After compiling with these, your eBPF ELF should have a `[BTF]` section (you can verify with `llvm-objdump -h` or `readelf -S`). Aya's loader (`aya::Bpf` or specifically the `aya-obj` crate) will detect and load this BTF section. The presence of BTF **does not change the fundamental format** (it's still an ELF64 relocatable of eBPF type), but it adds additional sections that Aya can use for relocations.

Aya uses BTF in a couple of ways: - **CO-RE field relocations:** If your eBPF code uses features like `#[allow(non_camel_case_types)] mod bindings { … }` to include kernel structure definitions (or uses `aya-tool`/`bindgen` to generate bindings), those references to kernel fields can be made relocatable. Aya's loader will, at load time, compare the BTF info encoded in your program against the target kernel's BTF info (retrieved via `Btf::from_sys_fs()` by default). It can then adjust your program's embedded offsets to match the actual kernel's struct layouts. This is how, for example, you can write an eBPF program on a kernel 5.10 header and still run it on 6.2 even if some struct grew or changed, as long as the relevant fields are present – BTF relocations handle that. - **Function relocations:** Aya also supports modern features like **global function calls and static variables** in eBPF [4]. BTF is involved in supporting **fentry** programs (which attach to function entry points) and may be used for resolving function call targets across kernels. However, note that Rust's compiler currently does not emit certain kinds of relocations (like

CO-RE field relocation records) as of early 2025 – there's an open issue about Rust not yet supporting BTF-defined relocations in eBPF programs. This means that while you can embed BTF type info (for debugging and manual use), you might not yet get automatic CO-RE behavior in pure Rust eBPF the same way C (libbpf) CO-RE works. Aya's loader is prepared to handle them once the compiler emits them, though.

From the user perspective, **including BTF is still valuable**: it allows you to use BPF skeleton features and get better error messages from the kernel (the verifier log can show type info), and it future-proofs your program for CO-RE. Aya does not *require* BTF, but if BTF is present and the kernel supports it, Aya will load the BTF info and make use of it (for example, Aya will attempt to load kernel BTF to assist in any relocations).

In summary, **BTF influences the emitted format only by adding extra sections** (BTF and BTF.ext). It doesn't change your code into "not eBPF" or anything – you still have eBPF bytecode in `.text` or analogous sections. Aya's inclusion of BTF is purely a supplementary feature that enhances portability. Always compile with debug info (at least for release builds you deploy) when using Aya, as this enables BTF. The overhead is minimal (BTF data is not loaded into the BPF VM's limited instruction count; it's only used at load time). The Aya homepage emphasizes that with BTF and static linking, Aya achieves a *"compile once, run everywhere"* solution.

6. Verifying the Output is Valid eBPF Bytecode

Once you have built your eBPF program (e.g., produced a file like `target/bpfel-unknown-none/debug/lock`), you should verify that it is indeed a loadable eBPF binary. There are a few ways to do this:

- **File type check:** Use the `file` command on the binary. It should report *"ELF 64-bit LSB relocatable, eBPF, version 1"* (or "BPF" as the architecture). If instead you see something like "current ar archive" or "LLVM IR bitcode", then it's not correct – it means you have .bc or .ll rather than a real ELF eBPF object. A correct output for an eBPF object is shown below:

```
$ file target/bpfel-unknown-none/debug/lock
target/bpfel-unknown-none/debug/lock: ELF 64-bit LSB relocatable, eBPF, version
1 (SYSV), not stripped
```

- **Disassemble to ensure bytecode is present:** Use `llvm-objdump -S` (or `objdump -d` if built with binutils supporting BPF) on the file. You should see human-readable eBPF assembly instructions. For example, lines of disassembly might look like `r1 = r6` or `call 25` etc., which correspond to eBPF opcodes. In the earlier example, the disassembly of section `xdp/myapp` is shown, confirming real instructions are there. If you only see data or nothing, something went wrong in the build.
- **Load with bpftool:** The ultimate test is loading the program into the kernel. You can use Aya's user-space to do this (as you would in the normal flow), or directly use `bpftool`. For instance: `sudo bpftool prog load target/bpfel-unknown-none/debug/lock /dev/null` (you have to specify a dummy map or pin location; bpftool syntax can be a bit involved). If the file is not a valid eBPF ELF, bpftool will error out. If it is valid, bpftool will load it (assuming your kernel allows unprivileged loads or you run as root) and you can then do `bpftool prog show` to see it. In the Aya example, after running the user-space with the eBPF loaded, `bpftool prog list` showed

the program with a specific ID, type (XDP, etc.), name, and that it's loaded and verified. This indicates the kernel recognized the program.

- **Check for BPF sections and symbols:** You can also inspect with `readelf -a` to ensure that sections like `.text` (or e.g. `kprobe/func_name`) exist and contain data, that a `"license"` section is present (kernel requires a license string in eBPF objects), and that relocations (if any) are resolved or present as expected.
- **Size and content sanity:** eBPF programs are usually quite small (a few KB). If your output ELF is extremely small (e.g. a few hundred bytes) or extremely large, that could indicate an issue (like it only has a symbol table and no code, or accidentally included lots of unnecessary stuff). For reference, a simple eBPF program compiled with Aya might be on the order of 3–5 KB in ELF size.

Given the question context, the main concern was distinguishing **LLVM bitcode vs actual eBPF**. The methods above will clearly differentiate them. LLVM bitcode files won't say "eBPF" in the file info and cannot be loaded by the kernel. A real eBPF ELF will. So if you follow the earlier steps (using `bpfel-unknown-none` target and `bpf-linker`), you should get a proper output. Always do a quick check with `file` and maybe a disassembler; this will save you from trying to debug an "Invalid ELF" error from the kernel later.

Example verification: After running `cargo xtask build-ebpf` in an Aya project, one can do:

```
$ ls -lh target/bpfel-unknown-none/debug/
... 3.5K Nov 6 22:24 myapp-ebpf
$ file target/bpfel-unknown-none/debug/myapp-ebpf
...: ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), not stripped
$ llvm-objdump -S target/bpfel-unknown-none/debug/myapp-ebpf | head -10
target/bpfel-unknown-none/debug/myapp-ebpf: file format elf64-bpf
Disassembly of section xdp/myapp:
0000000000000000 <myapp>:
   0:   bf 61 00 00 00 00 00 00     r1 = r6
   8:   18 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00     r2 = 0 ll
  18:   ... (etc) ...
```

This confirms the presence of eBPF instructions in the output.

7. Kernel Version Impact on Syscall Probes (e.g. `__x64_sys_connect`)

Kernel versions can significantly affect how you write eBPF **kprobe programs**, especially for system calls on x86_64. The symbol name you need to attach to and the context structure for arguments changed around Linux 5.2 to 5.3 and later in 5.7. Specifically for syscalls:

- On x86_64, since Linux 5.7, most syscalls were renamed internally to have an `__x64_` or `__x32_` prefix depending on the ABI, as part of cleaning up how pt_regs-based syscall functions are handled. For example, what used to be a function `sys_connect` (the syscall handler for `connect(2)`) became `__x64_sys_connect` for 64-bit callers, and `__sys_connect` became the common implementation that the x64 and other ABIs call into. The presence of these prefixes means that if you naively attach a kprobe to "sys_connect" on a modern x86_64 kernel, you might actually miss the call, because the actual function executing is `__x64_sys_connect` (which then calls `__sys_connect`).

- To make matters more confusing, hooking `__x64_sys_connect` vs `__sys_connect` can yield different results. As noted in a kernel patch, a kprobe on `__x64_sys_connect` might not capture arguments properly: *"For __sys_connect event, parameters can be fetched normally, but for __x64_sys_connect, parameters cannot be fetched."*. This is because `__x64_sys_connect` is a thin wrapper that prepares arguments (moving them from the legacy `pt_regs` into actual registers) before calling the real `__sys_connect`. If you attach a kprobe to the wrapper, the usual method of retrieving arguments (via `struct pt_regs` in BPF context) doesn't yield the intended values – they've been shuffled around. The kernel patch in July 2020 addressed this in tooling by adjusting how the kprobe event for `sys_connect` is handled.

- **Impact for Aya/Rust**: If you are writing an eBPF program to probe a syscall like `connect`, you need to be mindful of the kernel version:

- On new kernels (5.7+ for x86_64), you likely want to attach to `__x64_sys_connect` *or* handle the argument issue. In BPF, one way is to actually probe `__sys_connect` instead, which is after the argument setup (this might miss the top-level entry but still hits the actual implementation). The downside is on x86_64, `__sys_connect` is a static symbol not directly exported to kprobes (it might still work if you know the address or if kprobe events allow it by symbol name – this can depend on kernel config). Another way is to use **fentry/tracepoint** as described below.
- On slightly older kernels (5.2 <= v < 5.7), the BPF Type Format exists, but the syscalls might not have had the `__x64_` separation or it was in progress. Actually, the renaming happened in 5.6-5.7, so kernels 5.2, 5.3, 5.4 might still use `sys_connect` as the name (but possibly already had `__x64_sys_connect` depending on backports). The safe approach on those might have been to attach to `sys_connect` or check if `__x64_sys_connect` exists.

- On very old kernels (<5.2), BTF isn't available, but you can still use kprobes. You just attach to whatever the symbol is (likely `sys_connect` without prefix) – however, note that on *even older* (pre-4.17) kernels, the concept of `__x64_` didn't exist at all. The main challenge with old kernels is often other limitations (no BPF loop support, fewer helper functions), which is outside this syscall naming issue.

- **Using fentry (on 5.5+):** Starting with kernel 5.5, eBPF introduced *fentry* (and *fexit*) programs – these attach to function entry/exit with near-zero overhead using BTF information. Aya supports this via the `#[fentry]` attribute in eBPF code. If your target kernel is new enough, you could write an fentry program for `connect` which attaches *via BTF type info* to the function. The nice thing about fentry is you specify the function by name (and the BPF loader uses BTF to hook it). For example, `#[fentry(function = "inet_csk_accept")]` would hook that kernel function. In the case of `connect`, you might try `function = "__x64_sys_connect"` or `function = "sys_connect"`. Since BTF knows the function prototypes, it might allow you to use the more abstract name. It's worth checking the kernel's BTF data: on x86_64, the BTF type likely lists `__x64_sys_connect` (and possibly `__sys_connect`). If you attach to the wrong one, you'd face the same issue as kprobe. Likely attaching to `__sys_connect` via fentry would get you the arguments in registers (since that's the real implementation). The **minimum kernel for fentry is 5.5**, so any 5.5+ kernel (which includes all 6.x) can use it. Using fentry also bypasses the whole `pt_regs` argument extraction – you get a context like `FEntryContext` from which you can directly get function arguments via `ctx.arg(n)` in Aya, which should correspond to the real arguments if you hooked the implementation.

- **Using tracepoints:** Another portable way to catch syscalls is to use the existing tracepoint events. The kernel provides standard tracepoints for sys_enter and sys_exit of syscalls (e.g., `syscalls:sys_enter_connect` and `syscalls:sys_exit_connect`). Aya supports tracepoint programs (`#[tracepoint]` macro). These tracepoints abstract away the architecture differences – on any arch, `sys_enter_connect` will fire when a connect syscall is invoked, with a context that includes a common format (usually containing the syscall number and arguments). If you use a tracepoint, you don't worry about `__x64_` prefixes at all. The drawback is tracepoints have a bit more overhead and slightly different context structures, but for many purposes they are perfectly fine and stable across kernel versions. If your goal is simply to monitor or log syscalls like connect, a tracepoint program might be the simplest route to compatibility.

- **Syscall compatibility summary:** For a given Aya project that includes a kprobe on `connect`, you may need to handle multiple kernels:

- If targeting Linux 6.x on x86_64: you likely have to attach to `__x64_sys_connect` (the primary entry for 64-bit calls). Aya's kprobe attach API (in user-space) will accept a function name – you'd pass `"__x64_sys_connect"`. The eBPF program's section name would be `kprobe/__x64_sys_connect` if specified. However, remember the pitfall: at that point in execution, the `ProbeContext` (which uses pt_regs) might not directly give you the socket FD or addr parameter without adjustment. You might have to manually read the regs as the patch indicates (the kernel fix did something like reading RDI then using it to get RSI,RDX).
- If running the same eBPF on an older kernel (say 5.4), `__x64_sys_connect` symbol may not exist – the function could just be `sys_connect`. Your kprobe attach would fail if the symbol isn't found. One could work around this by detecting the kernel version at runtime in user-space and choosing the appropriate attach point. Aya doesn't do this automatically for you, but you can implement it (e.g., try one, catch error, try the other).

- Because of this hassle, the introduction of BTF and fentry is a boon: on 5.5+ you could use `#[fentry(function="connect")]` (if that works via BTF – one would need to confirm the exact name in BTF), and then you don't worry about the `__x64_` naming or pt_regs at all. The fentry will hook at the right spot and provide a context to get arguments easily. The Aya macro docs note that fentry and kprobe serve similar purposes but fentry has essentially *zero overhead* and is more efficient (plus no arch-specific quirks, aside from requiring BTF). Thus, for modern kernels, fentry is often the preferred mechanism.

- **Other kernel version considerations:** Beyond syscall naming, kernel versions affect eBPF in other ways. For example, kernel 5.2 was the first with BTF, 5.3 introduced support for BPF loops and function calls (before that, eBPF programs had to be linear and could only use tail calls for jumps). Kernel 5.4 added some new helper functions. Kernel 5.8 added the ring buffer map. By kernel 6.x, many new program types exist (struct ops, etc.) and the BPF verifier is much more capable. When writing Aya programs, you should be aware of the minimum kernel version your program needs. Aya tries to maintain backward compatibility – for instance, `bpf-linker` can emit loop-unrolled versions of your code if you ask, to target pre-5.3 kernels that don't support loops. Similarly, if you use a feature (like BPF spin locks in maps or new helpers), you must be on a kernel that supports it. In context, **for syscall probes specifically**, the key version-based difference is the naming and argument retrieval addressed above for 5.7+.

In practice, if your target deployment environment includes kernels both older and newer than 5.7, you might implement logic in your user-space loader: e.g., detect kernel version and choose to attach to `sys_connect` or `__x64_sys_connect`. Aya's high-level API might eventually abstract this (for example, the old BCC library would automatically add `__x64_` prefix for kprobes on x86_64 when needed – that logic came from the same kernel patch series). It's worth checking Aya's latest release notes; it may have introduced convenience for this. If not, it's something to handle manually.

Canonical example: A real-world scenario of this is discussed in the Linux kernel samples and mailing list. The kernel commit message we cited explains that initially a fix was made to auto-prefix `sys_` with `__x64_` for kprobe events on x86, but it broke `sys_connect` specifically. The resolution was to special-case how `connect`'s arguments are obtained. For eBPF developers, the takeaway is that `connect(2)` on x64 is a bit special when probing. The safest approach on new kernels is to probe `__sys_connect` or use tracepoints, or if you probe `__x64_sys_connect`, be prepared that `ctx.arg(0)` etc. might not be what you expect without adjustment.

In summary, **kernel 5.2+ vs 6.x differences** in the context of Aya are mainly about how you attach eBPF to certain hooks: - BTF (5.2+) available -> use CO-RE and new attach types (fentry) to avoid symbol issues. - Syscall renames (5.7) -> need to target the correct symbol for kprobes (and possibly adjust argument access). - Newer kernels (5.x to 6.x) progressively remove earlier limitations (so eBPF programs can be more complex on 6.x than on 4.x). If your `lock-ebpf` program uses features like maps, ensure the kernel supports them (e.g., *syscall kprobe programs* usually require at least 4.17 since that's when eBPF was allowed to probe syscalls, IIRC, and some syscalls might even have dedicated program types like `fmod_ret` etc., but Aya abstracts that).

Finally, whenever in doubt, consult Aya's documentation and examples for the specific probe you're writing. Many community examples (and Aya's own examples) illustrate how to handle these differences. For instance, a tutorial on writing a kprobe for TCP connect shows attaching to the internal `tcp_connect` function (which is stable across kernels) rather than the syscall. This is another strategy: sometimes hooking a higher-level kernel function yields more consistent results than the raw syscall. It all depends on what information you need.

References:

1. Alessandro Decina, *Adding BPF target support to the Rust compiler* – explains why a custom linker is needed for Rust eBPF.
2. `bpf-linker` README – usage instructions for Rust eBPF, including .cargo/config setup and BTF flags.
3. Aya documentation and blog examples – e.g., *Writing eBPF Programs with Rust Aya* (ebpf.top) which demonstrates building an eBPF ELF and checking it with `file`/`objdump`, and the Aya book's section on probes which discusses attaching to `tcp_connect` as an example.

4. Linux kernel commit message (Daniel T. Lee, 2020) – details the `__x64_sys_connect` vs `__sys_connect` issue on x86_64 and how kprobe events handle it. This provides insight into kernel version quirks for syscall probes.

---

[1] [2] Adding BPF target support to the Rust compiler | by Alessandro Decina | Medium
https://medium.com/@alessandrodecina/adding-bpf-target-support-to-the-rust-compiler-3ef113dbbd09

[3] [4] Home - Aya
https://aya-rs.dev/