BTF

Contents

1. BTF:

1.1. 🧠 What is BTF?

BTF = BPF Type Format.

It's a **metadata format** used by the ***eBPF subsystem*** in the Linux kernel to describe **types, structures, and debugging information** for BPF programs — very similar to how DWARF works for traditional compiled binaries.

Think of BTF as a **lightweight debugging/type info system** for BPF programs.

1.1.1. 🧬 Why is BTF Important?

Modern **eBPF** programs interact with kernel structures and data, which:

- can vary across kernel versions
- are not always visible at runtime

BTF enables these programs to:

1. Understand and access kernel data structures at runtime.
2. Be **portable** across kernel versions (via CO-RE).
3. Allow tools like `bpftool` or `bpftrace` to show symbols, fields, types.
4. Reduce the need for custom kernel headers.

1.1.2. ⚙️ Use Cases of BTF

Use Case	*Description*
CO-RE (Compile Once – Run Everywhere)	Enables portable eBPF programs that adapt to different kernel versions without recompilation.
bpftool introspection	Lets tools like **bpftool** show type info, structure layouts, and BPF maps/programs clearly.
bpftrace/DTrace-like tools	Allows high-level tracing tools to generate field-safe probes dynamically.
Verifier Assistance	Helps the BPF verifier with type safety, pointer validation, etc.

1.1.3. 🔧 How BTF Works

- The kernel may expose a file:

  ```
  /sys/kernel/btf/vmlinux
  ```

  Which contains BTF metadata for the running kernel.

- When you compile a BPF program with `clang`, you can generate BTF data with:

  ```
  clang -target bpf -g -O2 -g -Xclang -target-feature -Xclang +btf ...
  ```

- Tools like `bpftool`, `libbpf`, and `llvm` use this BTF section to:
  - read symbol types
  - access struct fields
  - perform CO-RE relocations

1.1.4. ✅ Summary

BTF = *Metadata for BPF Programs*

It allows:

- eBPF programs to understand kernel structure layouts
- runtime portability (CO-RE)
- user-friendly introspection and debugging
- better type safety in the kernel verifier

Without BTF, writing portable or introspectable eBPF programs would require building per-kernel versions or parsing headers manually — error-prone and fragile.

1.2. BTF Support (recompiling the kernel)?

Usually, yes — but it depends on your kernel version and distribution.

1.2.1. What kernel support is needed for BTF?

1. **Kernel Config Options**

To have full BTF support, the kernel must be built with these options enabled:

- `CONFIG_DEBUG_INFO_BTF=y` (This enables BTF debug info generation during kernel build.)
- `CONFIG_BPF_SYSCALL=y` (Enables BPF syscall support, usually already enabled.)
- `CONFIG_DEBUG_INFO=y` (Kernel debug info support; often needed together with BTF.)
- Optionally: `CONFIG_BPF_JIT_ALWAYS_ON` (for better BPF performance)

2. **Kernel Version**

- **Linux kernel 4.18+** started BTF support.
- **Kernel 5.3+** improved BTF and CO-RE support significantly.
- Many distributions backport BTF to their kernels.

3. **BTF Data Availability**

- After building the kernel with `CONFIG_DEBUG_INFO_BTF=y`,the **BTF data is embedded** inside the kernel img.
- This BTF data is typically exported at runtime as:

  `/sys/kernel/btf/vmlinux`
- Tools (like `bpftool`) read this file to get kernel type info.

1.2.2. What if your kernel doesn't have BTF?

- You **won't** have `/sys/kernel/btf/vmlinux`.
- You can **generate and supply your own BTF** files but it's complex.
- Some tools might still work but with reduced functionality.
- Many modern distros now ship kernels with BTF enabled by default.

1.2.3. Summary

Scenario	*BTF availability*
Kernel built with `CONFIG_DEBUG_INFO_BTF=y`	Full BTF support + `/sys/kernel/btf/vmlinux` present
Kernel without BTF config	No BTF, limited eBPF capabilities
Older kernels (before 4.18)	No BTF support

1.2.4. TL;DR

For proper BTF support, yes, you typically need to recompile the kernel with `CONFIG_DEBUG_INFO_BTF=y`. If you use a modern distro kernel (5.x+), it's often enabled already.

1.3. bpftool:

It's a modern eBPF tooling, its a tool for inspection and simple manipulation of eBPF programs and maps.

1.3.1. 🛠️ What is `bpftool`?

`bpftool` an official utility provided by the Linux kernel for *managing, introspecting, and debugging BPF objects*, including:

- BPF programs
- Maps
- Links
- BTF (BPF Type Format) metadata
- cgroup attachments
- netlink and tc hooks

It comes with the **kernel source** (under `tools/bpf/bpftool`) and also packaged by most major Linux distros.

1.3.2. 🔍 What is `bpftool` Used For?

Function	Description
`bpftool prog show`	Lists all loaded BPF programs
`bpftool map show`	Shows loaded BPF maps
`bpftool btf dump`	Dumps BTF (BPF Type Format) metadata
`bpftool feature probe`	Probes kernel for supported eBPF features
`bpftool net`	Inspects network stack hooks (tc, xdp)
`bpftool cgroup`	Shows BPF programs attached to cgroups

1.3.3. 🔬 How to Use `bpftool` to Read BTF Info

1. 🧠 Check if kernel BTF info is available

```
ls /sys/kernel/btf/vmlinux
```

If the file exists, the kernel exposes its BTF data (✅ good!).

2. 📖 Dump BTF types from the kernel

```
sudo bpftool btf dump file /sys/kernel/btf/vmlinux format c
```

This prints out the kernel's types in C-like syntax, e.g.:

```
struct task_struct {
    ...
    struct mm_struct *mm;
    int pid;
    ...
};
```

You can also filter specific types:

```
sudo bpftool btf dump file /sys/kernel/btf/vmlinux format c | grep -A 20 "struct
task_struct"
```

3. 🧪 Inspect BTF in an ELF/BPF Object File

If you compiled a BPF program with `clang -g -target bpf`, the BTF info is embedded in the object file:

```
bpftool btf dump file myprog.o format c
```

This is useful for:

- Debugging structure layouts
- Verifying CO-RE compatibility
- Understanding what the verifier sees

1.3.4. 💡 Practical Example

```
bpftool btf dump file /sys/kernel/btf/vmlinux format raw
```

This dumps the BTF types in raw format (for scripting or analysis).

Or for human-readable output:

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c
```

–

1.3.5. 🧰 Installing `bpftool`

- Ubuntu/Debian:

```
sudo apt install bpftool
```

- On Fedora:

```
sudo dnf install bpftool
```

Build from kernel source:

```
cd /usr/src/linux/tools/bpf/bpftool
make
sudo make install
```

1.3.6. ✅ Summary

Command	Purpose
`bpftool btf dump file /sys/kernel/btf/vmlinux format c`	View kernel BTF types
`bpftool btf dump file prog.o format c`	Inspect BTF in a BPF object
`bpftool prog/map/show`	List loaded BPF programs or maps
`bpftool feature probe`	See kernel eBPF/BTF capabilities

1.4. Example how bpftool and bpftrace

1.4.1. 🧪 1. CO-RE BPF Program Example (Trace Process Execs) real example of a CO-RE (Compile Once – Run Everywhere)

BPF program and how **BTF** shows up in `bpftool` and `bpftrace`.

This example traces `execve` calls using eBPF and accesses kernel struct fields via BTF.

1.4.2. 🔧 Requirements

- Kernel 5.3+ with BTF enabled (`/sys/kernel/btf/vmlinux` exists)
- `clang`, `llvm`, `bpftool`, `libbpf-dev`
- BPF CO-RE headers (from kernel or `linux-headers`)

1.4.3. 📁 File: `trace_exec.c`

```c
// SPDX-License-Identifier: GPL-2.0
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>

char LICENSE[] SEC("license") = "GPL";

SEC("tracepoint/syscalls/sys_enter_execve")
int trace_exec(struct trace_event_raw_sys_enter* ctx)
{
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    pid_t pid = BPF_CORE_READ(task, pid);
    char comm[16];
    BPF_CORE_READ_STR(&comm, task, comm);

    bpf_printk("exec: %d %s\n", pid, comm);
    return 0;
}
```

This uses:

- `BPF_CORE_READ()` – to safely access `task_struct::pid` across kernels.
- **No kernel headers** are needed — it reads types from **BTF** at runtime.

1.4.4. 🏗 Compile with BTF

```
clang -g -O2 -target bpf \
  -D__TARGET_ARCH_x86 \
  -I/path/to/vmlinux.h \
  -I. \
  -c trace_exec.c -o trace_exec.o
```

You must have `vmlinux.h` (BTF-generated header). You can generate it using `bpftool`:

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

Or:

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c | pahole -J - > vmlinux.h
```

1.4.5. 🔎 2. How BTF Shows Up in `bpftool`

🧩 Inspect BTF info in a BPF object:

```
bpftool btf dump file trace_exec.o format c
```

You'll see:

```
struct task_struct {
    int pid;
    char comm[16];
    ...
};
```

This proves the object has **embedded BTF metadata**.

1.4.6. 🔍 View kernel BTF types:

```
bpftool btf dump file /sys/kernel/btf/vmlinux format c | grep -A 5 'task_struct'
```

You'll see the actual kernel `task_struct` layout, used by CO-RE.

–

1.4.7. 🧯 3. Run It with libbpf-based loader or `bpftool`

If using `bpftool` to load:

```
sudo bpftool prog load trace_exec.o /sys/fs/bpf/trace_exec
```

Attach it:

```
sudo bpftool prog attach /sys/fs/bpf/trace_exec \
    tracepoint syscalls sys_enter_execve
```

Then check the logs:

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

You'll see:

```
exec: 12345 bash
exec: 12346 ls
```

1.5. 🧠 4. BTF in `bpftrace`

1.5.0.1. Example:

```
sudo bpftrace -e 'kprobe:do_exit { printf("Exiting PID: %d\n", pid); }'
```

`bpftrace` automatically uses **BTF** from `/sys/kernel/btf/vmlinux` to know what `do_exit` is and how to access its parameters or fields.

It enables zero-effort tracing using BTF introspection.

1.5.1. ✅ Summary

Tool	How BTF Helps
`bpftool`	Dumps struct/type info from BPF objs or kernel
`bpftrace`	Auto-detects kernel struct layouts via BTF
`libbpf`	Enables BPF CO-RE with BTF relocations
`clang`	Emits `.BTF` section in object files

:

1.6. Working Makefile or how to set up a `vmlinux.h` generation script?

Everything you need to **compile and load a BPF CO-RE program**, including a **`Makefile`** and a way to **generate `vmlinux.h`** automatically using `bpftool`.

✅ Folder Structure

```
bpf-core-trace/
├── Makefile
├── trace_exec.c
├── vmlinux.h          <-- auto-generated
```

🧾 1. `trace_exec.c` (same as before)

```c
// SPDX-License-Identifier: GPL-2.0
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>

char LICENSE[] SEC("license") = "GPL";

SEC("tracepoint/syscalls/sys_enter_execve")
int trace_exec(struct trace_event_raw_sys_enter* ctx)
{
    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    pid_t pid = BPF_CORE_READ(task, pid);
    char comm[16];
    BPF_CORE_READ_STR(&comm, task, comm);

    bpf_printk("exec: %d %s\n", pid, comm);
    return 0;
}
```

🛠️ 2. `Makefile`

```makefile
CLANG ?= clang
BPFTOOL ?= bpftool
BPF_OBJ = trace_exec.o
SRC = trace_exec.c

.PHONY: all clean

all: $(BPF_OBJ)

# Generate vmlinux.h from kernel BTF
vmlinux.h:
	@if [ ! -e /sys/kernel/btf/vmlinux ]; then \
	  echo "ERROR: Kernel BTF not found at /sys/kernel/btf/vmlinux"; \
	  exit 1; \
	fi
	@echo "[*] Generating vmlinux.h from BTF..."
	$(BPFTOOL) btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h

$(BPF_OBJ): $(SRC) vmlinux.h
	@echo "[*] Compiling BPF program..."
	$(CLANG) -g -O2 -target bpf \
```

```
    -D__TARGET_ARCH_$(shell uname -m | sed 's/x86_64/x86/') \
    -I. \
    -c $(SRC) -o $(BPF_OBJ)

clean:
  rm -f $(BPF_OBJ) vmlinux.h
```

🧪 3. Build the BPF Program

```
make
```

This will:

- Check for `/sys/kernel/btf/vmlinux`
- Generate `vmlinux.h`
- Compile `trace_exec.o` with BTF info

🔗 4. Load & Attach the BPF Program

You can load and attach it using `bpftool` or a loader like [`libbpf`](https://github.com/libbpf/libbpf).

Using `bpftool`:

```
# Load the BPF program
sudo bpftool prog load trace_exec.o /sys/fs/bpf/trace_exec type tracepoint

# Attach it to tracepoint
sudo bpftool prog attach name trace_exec \
    tracepoint syscalls:sys_enter_execve
```

📜 5. See the Output

Read from `trace_pipe`:

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

You'll see something like:

```
exec: 2345 bash
exec: 2346 ls
```

1.6.1. 📎 Notes

- If you get `vmlinux.h: file not found` — ensure `/sys/kernel/btf/vmlinux` exists.
- Requires: kernel 5.4+ (for stable CO-RE), `bpftool`, `clang`, `llvm`, and BPF headers.

2. Aya Rust based eBPF Development:

Connecting the above with **Aya** a **Rust-based eBPF development**.

Aya is a modern, pure-Rust framework for writing and loading eBPF programs **without relying on libbpf or C code**.

2.1. 🧠 What is Aya?

Aya is:

- A **Rust framework** for writing eBPF programs (like `trace_exec`) in Rust
- Provides **CO-RE** (Compile Once – Run Everywhere) support using **BTF**
- Does **not require libbpf or clang/LLVM** (only Rust toolchain)
- Works on **modern Linux kernels** (with BTF)

> It bridges **Rust ↔ Kernel BPF**, using BTF for safe type access like the `BPF_CORE_READ()` macro in C, but with type-safe Rust code.

2.2. 📌 How the Above Concepts Fit with Aya

Concept	In C (`libbpf`)	In Rust (Aya)
Program language	C	Rust
Loader	`bpftool` / custom C	Rust user-space app with Aya runtime
Kernel struct access	`BPF_CORE_READ()` + `vmlinux.h`	Aya auto-generates BTF access
BTF requirement	`/sys/kernel/btf/vmlinux`	✅ Required for CO-RE (via BTF)
Program sections	`SEC("tracepoint/...")`	`#[map]`, `#[tracepoint]`, etc. macros
Compilation	`clang -target bpf`	`cargo xtask` (or `cargo bpf`)
Safety	Unsafe C	Safe or unsafe Rust

2.3. 🔁 CO-RE in Aya

Just like in the C example where we used:

```
BPF_CORE_READ(task, pid);
```

Aya lets you **generate and reference BTF types from Rust** using macros like:

```
use aya_bpf::bindings::task_struct;

#[tracepoint(name = "sys_enter_execve")]
pub fn trace_exec(ctx: TracePointContext) -> i32 {
    unsafe {
        let task: *const task_struct = bpf_get_current_task();
        let pid = (*task).pid;
        // ... log or trace comm
    }
    0
}
```

Aya uses **bindgen** and **BTF introspection** behind the scenes to safely access those fields.

2.3.1. 🛠️ How Aya Builds and Loads eBPF

Aya typically uses a split-project layout:

```
project/
├── xtask/              # Loader/CLI
├── bpf/                # eBPF programs in Rust
│   └── src/
│       └── trace_exec.rs
├── Cargo.toml
```

2.3.1.1. BPF Program (Rust) → CO-RE ELF

The eBPF program is compiled using:

```
cargo xtask build-ebpf
```

It embeds:

- `.BTF` section for CO-RE
- Symbol section (like in `trace_exec.o`)
- Maps and program sections

2.3.1.2. Loader → Loads BPF into kernel

```
let mut bpf = Bpf::load_file("target/bpf/trace_exec.o")?;
program.attach()?;
```

The loader handles:

- BTF relocations
- Program/map loading
- Attachments to hooks

2.3.2. 🔍 Checking BTF in Aya-generated ELF

You can inspect BTF in an Aya-generated `.o` with `bpftool`, just like C:

```
bpftool btf dump file target/bpf/trace_exec.o format c
```

You'll see the Rust-generated types used in the BPF program — verified via BTF.

2.3.3. ✅ Summary: How it All Fits Together with Aya

Component	Role
BTF	Provides kernel type metadata
CO-RE	Lets programs access structs like `task_struct` across kernel versions
Aya	Rust framework for writing + loading BPF
`trace_exec.c`	Equivalent written in Aya's Rust DSL
`vmlinux.h`	Not used directly — Aya uses BTF instead
`bpftool`	Still useful for inspecting objects

2.4. Example Rust/Aya version of `trace_exec`

Minimal Aya Rust CO-RE eBPF example that traces `execve` syscalls just like the C example, with instructions and a sample project structure.

Aya CO-RE Example: Trace execve

Step 1. Project structure

```
aya-trace-exec/
├── bpf/
│   ├── Cargo.toml
│   └── src/
│       └── trace_exec.rs
├── Cargo.toml
└── src/
    └── main.rs       # Loader program
```

Step 2. `bpf/Cargo.toml` – eBPF program crate

```toml
[package]
name = "trace_exec"
version = "0.1.0"
edition = "2021"

[dependencies]
aya-bpf = { version = "0.19", features = ["tracepoint"] }
aya-log = "0.19"

[lib]
crate-type = ["cdylib"]
```

–

Step 3. `bpf/src/trace_exec.rs` – BPF program

```rust
#![no_std]
#![no_main]

use aya_bpf::{
    bindings::task_struct,
    macros::tracepoint,
    programs::TracePointContext,
    helpers::bpf_get_current_task,
};
use aya_log_ebpf::info;

#[tracepoint(name = "sys_enter_execve")]
pub fn trace_exec(ctx: TracePointContext) -> u32 {
    match unsafe { try_trace_exec(ctx) } {
        Ok(ret) => ret,
        Err(_) => 1,
    }
}

unsafe fn try_trace_exec(_ctx: TracePointContext) -> Result<u32, ()> {
    let task = bpf_get_current_task();
    if task.is_null() {
        return Err(());
    }
```

```rust
    let pid = (*task).pid;
    // comm is fixed-size 16 bytes char array
    let comm = (*task).comm;

    // Log the exec call with PID and comm (command name)
    // This shows up in `trace_pipe` if tracing is enabled
    info!(&format!("exec: {} {:?}", pid, comm));

    Ok(0)
}
```

Step 4. Top-level `Cargo.toml`

```toml
[workspace]
members = [
    "bpf",
]
```

Step 5. `src/main.rs` — Loader

```rust
use aya::{Bpf, programs::TracePoint, util::online_cpus};
use aya::maps::perf::PerfEventArray;
use aya::util::online_cpus;
use std::convert::TryInto;
use std::error::Error;
use std::sync::Arc;
use tokio::{signal, task};

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    // Load the compiled eBPF program
    let mut bpf = Bpf::load_file("bpf/target/bpfel-unknown-none/release/
trace_exec.o")?;

    // Attach tracepoint
    let program: &mut TracePoint = bpf.program_mut("trace_exec").unwrap().try_into()?;
    program.load()?;
    program.attach("syscalls", "sys_enter_execve")?;

    println!("Attached to sys_enter_execve tracepoint. Waiting for events...");

    // Keep the program running
    signal::ctrl_c().await?;
    println!("Exiting...");

    Ok(())
}
```

Step 6. Build instructions

You need Rust nightly with `cargo` and `rustup` set up.

- Install Rust and aya-bpf dependencies

```bash
rustup target add bpfel-unknown-none
cargo install cargo-binutils llvm-tools-preview
```

- Build the BPF program

```bash
cd bpf
cargo build --release --target bpfel-unknown-none
```

- Build and run the loader

```bash
cd ..
cargo run --release
```

Step 7. View Output

Open a separate terminal:

```bash
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

You should see lines like:

```
exec: 12345 [bash, ...]
```

Notes:

- Aya uses BTF automatically — no need for `vmlinux.h`
- The `trace_exec.rs` uses safe Rust with some `unsafe` block to dereference kernel pointers
- This example uses `aya-log` for simple `bpf_printk` logging
- Aya supports other program types too — kprobes, uprobes, xdp, etc.

3. Solana:

- Solana smart contracts are mostly written in Rust, often compiled into BPF Bytecode.
- Experience writing CO-RE BPF programs in Rust (eg: Aya) is highly transferable to Solana contract devl.
- Note As Solana BPF is a custom, sandboxed VM for executing blockchain programs and Solana BPF VM does not use BTF and similar kernel debugging infrastructure.
- Solana programs have deterministic execution, limited compute budgets and no kernel interaction.