

BPF FD Loader: Integrity-Verified BPF on Android

Enabling On-Demand BPF for Android Platform & Vendors

Siddharth Nayyar <sidnayyar@google.com>

Thiébaud Weksteen <tweek@google.com>



The Status Quo: BPF Confined to Early Boot

Android's current BPF architecture relies on a synchronous, early-boot userspace loading chain.

Synchronous Boot Chain

- **Chained Execution**

Early boot runs a sequential chain of userspace loaders across domains:

- NetBpfLoad (Mainline)
- system `bpfloader`
- vendor `bpfloader`

- **Userspace Parsing**

Loaders use `libbpf` to parse ELF sections, create maps, and perform CO-RE relocations in userspace memory, followed by `sys_bpf()` call.

- **Permanent Termination**

Once early-boot loading completes, the loader process exits permanently.

Post-Boot Lockdown

- **Exclusive SELinux Domain**

Android's SELinux policy strictly grants `BPF_PROG_LOAD` permission only to the `bpfloader` domain.

- **No Active Caller Post-Boot**

Because the `bpfloader` process terminates after early boot, no running process retains permission to load BPF programs.

- **Unreachable Syscall**

Beyond early boot, there is effectively no mechanism in userspace to invoke `sys_bpf(BPF_PROG_LOAD)`.

Key Consequences

- ✓ **Minimized Attack Window**

Loading all programs synchronously during early boot minimizes exploitation window before the rest of the system starts (like network / user processes).

- ✗ **Platform Rigidity & Memory Cost**

Platform services cannot load BPF on-demand; all programs and maps must load unconditionally at boot, permanently consuming RAM.

- ✗ **Vendor Lockout**

There is no official support for OEMs and SoC vendors to load or manage their own BPF programs.



Android's Security Posture & Other Attempts

Why relaxing SELinux, signed BPF, or raw ELF loading fall short on Android.

SELinux Policy strictness

- **Grant Post-Boot `sys_bpf()`?**

What if we loosen SELinux policy to allow privileged userspace daemons to invoke `sys_bpf(BPF_PROG_LOAD)` post-boot?

- **Mutable Userspace Memory**

Userspace loaders parse ELF's and relocate instructions in RAM, breaking the chain of trust to the original file.

- **Violates Verified Boot**

The kernel cannot trust bytecode buffers prepared in userspace; compromised daemons could inject malicious bytecode.

Pitfalls of Signed BPF

- **In-Kernel Crypto Flaws**

Relies on legacy in-tree RSA and complex [ASN.1/PKCS#7](#) parsers – fragile and bloats kernel by ~200 KB.

- **GKI Key Management**

In a decentralized ecosystem with dozens of OEMs and SoC vendors, managing signing keyrings in GKI is intractable.

- **Build & Signing Complexity**

Requires integrating dedicated cryptographic signing pipelines and key rotation across every vendor and platform build.

Raw ELF Loading via FD

- **Earlier Upstream Attempt**

Previous Android proposal (LSFMM/BPF '24) passed standard BPF ELF file descriptors directly to the kernel.

- **Informal ELF Conventions**

Upstream maintainers rejected freezing [libbpf](#)'s informal ELF section conventions into a stable kernel UAPI.

- **In-Kernel Complexity**

Pushing full ELF parsing and CO-RE relocation logic into kernel space was deemed unmaintainable.



The Blueprint: `finit_module()` as the Gold Standard

Android solved secure runtime code loading for kernel modules. Can we apply this model to BPF?

Verified Storage Model

- **Hardware Root of Trust**
Android Verified Boot (AVB) cryptographically seals `/system` and `/vendor` partitions.
- **Block-Level Integrity**
`dm-verity` continuously verifies read-only partitions directly in the kernel block layer.
- **Built-In Authenticity**
The kernel already has cryptographic certainty over any file residing on verified storage.

Kernel Module Precedent

- **File Descriptor Loading**
Android loads `.ko` modules via `finit_module()`, passing a file descriptor instead of a memory buffer.
- **Direct Kernel Reading**
Kernel uses `kernel_read_file()` to read module bytes directly from verified storage into kernel space.
- **LSM Policy Enforcement**
SELinux hooks verify that the file descriptor points to an authorized `dm-verity` partition.

The Blueprint for BPF

- **Identical Threat Model**
Like kernel modules, BPF programs execute in kernel space and require strict provenance guarantees.
- **Eliminating TOCTOU attacks**
Reading BPF objects directly from an authenticated file descriptor prevents userspace memory tampering.
- **! The Hurdle**
How can the kernel load a BPF file directly without parsing complex ELF sections or CO-RE relocations?



BPF Light Skeletons: The Self-Sufficient Artifact

Replicating `finit_module()` requires eliminating `libbpf` from the runtime workflow.

The Missing Kernel Linker

- **Kernel Modules (`.ko`)**

The kernel has an in-tree ELF linker that directly resolves and links `.ko` files passed via `finit_module(fd)`.

- **BPF Has No Kernel Linker**

The kernel only accepts raw bytecode; ELF parsing, map creation, and CO-RE relocations normally require `libbpf`.

- **The Dilemma**

Raw BPF ELF files cannot be handed directly to the kernel for loading.

Compile-Time Generation

- **Recording `libbpf` Actions**

At build time, `bpftool gen skeleton -L` runs `libbpf` in dry-run mode, recording every syscall step.

- **Encapsulated Syscall Program**

Emits a wrapping BPF program (`BPF_PROG_TYPE_SYSCALL`) containing map creation and CO-RE relocation logic.

- **Static `libbpf` Inside BPF**

Effectively embeds a static, program-specific version of `libbpf` directly inside the compiled artifact.

Self-Sufficient & Isolated

- ✓ **In-Kernel Execution**

When executed, the loader program performs CO-RE relocations and loads the target BPF programs entirely in kernel space.

- ✓ **Zero Runtime `libbpf` Dependency**

Userspace no longer needs to parse ELF files or construct bytecode buffers in memory.

- ✓ **`libbpf` API / ABI Isolation**

Because loading logic is self-contained, BPF artifacts are completely insulated from platform `libbpf` version skew.



The Solution: The BPF FD Loader Architecture

Bridging Light Skeleton Loader ELF FDs directly to `kern_sys_bpf()`.

The Ingress Device

- **`/dev/android_bpfloader`**

A minimal GKI misc character device driver that sidesteps the `sys_bpf()` impasse.

- **Light Skeleton Loader FD**

Userspace opens the verified light skeleton loader ELF (`.bpf`) and passes its FD via `ioctl`.

- **Zero Bytecode Handling**

Userspace never parses, allocates, or touches bytecode buffers in process RAM.

Zero Tampering

- **Direct Kernel Space Ingestion**

Driver invokes `kernel_read_file()` to pull the light skeleton ELF directly into kernel memory.

- **Eliminates TOCTOU attacks**

Bypasses mutable process memory entirely; the kernel operates on the authentic disk image.

- **In-Kernel Validation**

Validates ELF headers, verifies section bounds, and enforces image size limits (e.g. 8 MB).

Reusing `kern_sys_bpf()`

- **No New BPF Engine**

Driver does not reinvent loading – reuses existing in-kernel functionality (e.g. `kern_sys_bpf()`).

- **Executes the Loader**

`kern_sys_bpf()` verifies & runs the `BPF_PROG_TYPE_SYSCALL` loader from the ELF to create maps and programs.

- **Returns FDs to Caller**

Instantiated map and program FDs are installed into caller's FD table and returned.



Security Enforcement: Defense in Depth

Guaranteeing provenance and strict containment without exposing raw `sys_bpf()`.

✓ Filesystem Provenance

- **Kernel-Enforced Origin**
`kernel_read_file()` inspects the backing file at the VFS layer before reading any bytes.
- **Strict Partition Lockdown**
SELinux strictly forbids loading code from any location other than verified read-only partitions.
- **Cryptographic Verification**
Files are sealed by `dm-verity` rooted in hardware AVB; mutable paths are unconditionally blocked.

✓ Granular SELinux Policy

- **Isolated Device Label**
`/dev/android_bpfloader` is assigned its own dedicated SELinux label, completely isolating it from generic device nodes.
- **Targeted Authorization**
Access is granted solely to authorized userspace loaders.
- **Kernel-Only Program Loading**
Permanently blocks `BPF_PROG_LOAD` across all userspace domains, ensuring only the kernel can load BPF bytecode.

✓ Kernel-Level Guardrails

- **Capability Verification**
The driver strictly checks `bpf_capable()` before servicing any request.
- **Hardened Device Permissions**
Device node is created with mode `0600` (root-restricted).
- **Sanity Validation**
In-kernel validation of ELF header magic, section bounds, and image size ceiling (e.g. `SZ_8M`).



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Security Guarantee: Code provenance is proven at the VFS layer, while SELinux guarantees zero userspace exposure to raw `BPF_PROG_LOAD`.

End-to-End Lifecycle: From Source to Execution

How an integrity-verified BPF program travels from build to runtime execution.



1

Build & Package

Compile BPF bytecode



Generate Iskel loader



Place in **dm-verity**
sealed partition



2

Runtime Ingress

Userspace opens BPF loader FD



Pass FD to kernel space via
Android driver's **ioctl**

★ Userspace can demand
BPF program loading
while **sys_bpf()** remains
locked down.



3

Kernel Execution

Driver pulls ELF from
dm-verity sealed partition



BPF loader maps and prog
are allocated and executed

★ Target BPF program
relocation and loading
happens entirely in
kernel space.



4

Handoff & Attach

Driver installs map and prog
FDs into caller's FD table



Caller attaches program



Caller pins maps and progs



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Key Benefits, Technical Roadmap & Upstream Alignment

Unlocking on-demand BPF across Android while aligning with upstream Linux.

Upstream Alignment

- **Shared Light Skeleton Foundation**

Our packaging format aligns 100% with upstream signed BPF (Linux 7.2); we proposed `BPF_LOADER_LOAD_FD` upstream to load light skeletons directly via file descriptor.

- **Bridging the Gap via GKI Driver**

Upstream alternatives (`dm-verity` LSM helpers + `PKCS#7`) still require `X.509` crypto or trust userland RAM; our GKI driver bridges this gap cleanly while dialogue continues.

Key Benefits

- **Official Vendor BPF Support**

Provides OEMs and SoC vendors an officially supported, integrity-verified path to ship BPF.

- **Preserving Security Posture**

Maintains strict lockdown on raw `BPF_PROG_LOAD` in userspace, keeping Android's Verified Boot boundary intact.

- **On-Demand BPF Loading**

Unlocks loading BPF programs on demand, rather than locking the system to a fixed set of programs post-boot.

- **Zero Boot Memory Bloat**

Eliminates permanently resident early-boot probes and maps, preserving baseline RAM for user applications.

Technical Roadmap

- **Closing Light Skeleton Feature Gaps**

Upstreaming missing primitives to unlock advanced features like `sched_ext`:

- `STRUCT_OPS` support
- `kfunc` versioning using flavor suffixes

- **Open Questions**

→ The high-level `kern_sys_bpf()` interface creates userspace accessible FDs for maps and programs. How should we prevent TOCTOU attacks? Can we use a different kernel interface or abstraction?

→ Android offers ABI stability and compatibility for the kernel module interface. What should be the ABI posture for BTF and `kfuncs` across GKI releases?



LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Thank You

Questions & Discussion

Siddharth Nayyar (Presenter)
Thiébaud Weksteen (Co-Author)



**LINUX
PLUMBERS
CONFERENCE**

PRAGUE, CZECHIA / OCT. 5-7, 2026