# A Deep Dive Into eBPF Program Loader

Cong Wang xiyou.wangcong@gmail.com
Linux Kernel Maintainer

*Open Source Summit North America, 2025*

# Agenda

1. **eBPF Objects & Syscall Interface**
2. **Four-Phase Loading Pipeline**
3. **BTF Type System & CO-RE**
4. **Kernel Verifier Interaction**
5. **Program Signing Challenges**

# eBPF Code Example

```
// Map definition (demo)
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 1024);
    __type(key, int);
    __type(value, struct event);
} events SEC(".maps");

// Program definition
SEC("kprobe/sys_open")
int trace_open(struct pt_regs *ctx) {
    bpf_printk("Hello world!");
    return 0;
}
```

# eBPF Objects: Programs and Maps

**1. Programs**

- Executable bytecode
- Verified by kernel
- Attached to kernel hooks
- Type-specific (XDP, kprobe, etc.)

**2. Maps**

- Data structures for storage
- Shared between programs/userspace
- Various types (hash, array, etc.)
- Persistent across program runs

# eBPF Objects: BTF and Links

**3. BTF (BPF Type Format)**

- Type information metadata
- Enables CO-RE and debugging
- Describes program/map structures
- Kernel and userspace type matching

**4. Links**

- Connection between program & hook
- Manages attachment lifecycle
- Automatic cleanup on process exit
- Reference counting for sharing

# eBPF Object Creation via Syscall

```
// Create a map
map_fd = bpf(BPF_MAP_CREATE, &map_attr, sizeof(map_attr));
// Load a program
prog_fd = bpf(BPF_PROG_LOAD, &prog_attr, sizeof(prog_attr));
// Load BTF type information
btf_fd = bpf(BPF_BTF_LOAD, &btf_attr, sizeof(btf_attr));
// Create attachment link
link_fd = bpf(BPF_LINK_CREATE, &link_attr, sizeof(link_attr));
```

**Key Insight**: File descriptors (fd) are the kernel's handle to loaded objects

# File Descriptor Management

- Each object gets a unique file descriptor
- FDs enable object sharing and persistence
- Objects destroyed when all FDs closed
- Can be pinned to filesystem for persistence
- **Challenge:** FDs are unpredictable at compile time

# From Source Code to ELF Binary

**Compilation Process:**

```
# Compile eBPF C source to ELF object file
clang -target bpf -O2 -c program.c -o program.o
```

**What happens during compilation:**

- eBPF C source code → eBPF bytecode instructions
- Metadata (maps, BTF) → ELF sections
- Result: ELF binary containing **only** eBPF programs and metadata
- ibbpf is **not** linked into the eBPF binary
- libbpf is a userspace library linked into **your** loader

# ELF Code and Data Sections

- `SEC("type/name")` - Main program bytecode
- `.text` - Subprogram/helper function instructions
- `.rodata` - Read-only data (strings, constants)
- `.data` - Initialized global variables
- `.bss` - Uninitialized global variables

# ELF Metadata Sections

- `.maps` - Map definitions and attributes
- `.BTF` - Type information (structs, functions)
- `.BTF.ext` - Source line info, function info
- `.rel*` - Relocation information

# ELF to Kernel Object

- Step 1: ELF Parsing
  - `obj = bpf_object__open("program.o");`
  - Parse ELF file, extracts sections, symbols, relocations
- Step 2: Object Preparation
  - `bpf_object__prepare(obj);`
  - Resolves symbols, creates maps, processes relocations
- Step 3: Program Loading
  - `bpf_object__load(obj);`
  - Each program loaded and verified for safety and correctness
- Step 4: Attachment and Execution

# The Complete Loading Process

```
┌──────────────┐     ┌───────────────┐     ┌───────────────────┐     ┌──────────────────┐
│ ELF Object   │ ──▶│    libbpf     │ ──▶│ Kernel Objects    │ ──▶│ Running Program  │
│  .o file     │     │  Processing   │     │ (via bpf())       │     │ (attached)       │
└──────────────┘     └───────────────┘     └───────────────────┘     └──────────────────┘
```

**Key Insight:** libbpf acts as a sophisticated translator between static ELF representation and dynamic kernel objects

# Core Data Structures: `struct bpf_object`

```
enum bpf_object_state {
    OBJ_OPEN,         // ELF parsed, sections identified
    OBJ_PREPARED,     // Maps created, relocations done
    OBJ_LOADED        // Programs loaded to kernel
};

struct bpf_object {
    char name[BPF_OBJ_NAME_LEN];        // Object name
    char license[64];                   // License string
    enum bpf_object_state state;        // Loading state

    struct bpf_program *programs;       // Program array
    struct bpf_map *maps;               // Map array
    struct btf *btf;                    // Object BTF
    struct btf *btf_vmlinux;           // Kernel BTF

    int *fd_array;                      // File descriptor storage
    size_t fd_array_cnt;                // Number of FDs
};
```

# Core Data Structures: Programs & Maps

## Program Representation

```
struct bpf_program {
    char *name;                             // Program name
    enum bpf_prog_type type;                // XDP, kprobe, etc
    struct bpf_insn *insns;                 // Instructions
    size_t insns_cnt;                       // Instruction count

    int fd;                                 // Kernel FD after load
    struct reloc_desc *reloc_desc;          // Relocations
    int nr_reloc;                           // Relocation count
};
```

## Map Definition

```
struct bpf_map {
    char *name;                            // Map name
    enum bpf_map_type type;                // HASH, ARRAY, etc
    __u32 key_size, value_size;            // Entry sizes
    __u32 max_entries;                     // Capacity

    int fd;                                // Kernel FD after create
    __u32 btf_key_type_id;                 // BTF type IDs
    __u32 btf_value_type_id;
};
```

# Phase 1 - Discovery

### ELF Section Processing:

```
bpf_object__elf_collect() // Parse ELF sections (.text, .maps, .BTF)
bpf_object__add_programs() // Extract program instructions from .text
bpf_object__init_user_btf_maps() // Parse map definitions from .maps
```

### BTF and Metadata:

```
bpf_object__init_btf() // Load BTF type information
```

**Output**: Parsed object with identified sections, programs, and maps

# Phase 2 - Resolution

## Kernel Interaction Setup:

```
bpf_object_prepare_token() // Prepare for kernel interaction
btf__load_vmlinux_btf() // Load kernel BTF for type matching
```

## Symbol Resolution & Relocations:

```
bpf_object__resolve_externs() // Resolve external symbols (kernel config, functions)
bpf_object__relocate() // Process all relocations
```

## Map Creation

```
bpf_object__create_maps() // Create eBPF maps with BPF_MAP_CREATE
```

**Output**: Fully resolved objects ready for kernel loading

# Phase 3 - Kernel Interaction

**Program Loading Loop:**

```
bpf_object__load_progs(obj, extra_log_level) {
    for (i = 0; i < obj->nr_programs; i++) {
        bpf_object_load_prog(obj, prog, prog->insns, prog->insns_cnt,
                                       obj->license, obj->kern_version, &prog->fd);
    }
}
```

**Verifier Interaction:**

```
fixup_verifier_log() // Post-process verifier log to improve error descriptions
```

**Output**: Loaded programs with kernel file descriptors

# Phase 4 - Attachment

**Program Attachment Methods:**

```
// Auto-attachment based on SEC() definition
bpf_object__attach_skeleton(skel);
// Manual attachment for specific hooks
link = bpf_program__attach_kprobe(prog, false, "sys_open");
link = bpf_program__attach_xdp(prog, ifindex);
link = bpf_program__attach_cgroup(prog, cgroup_fd);
```

**Execution Context:** Program runs when kernel events trigger the attached hook

# Link Management

- Links represent the attachment relationship
- Can be pinned to filesystem for persistence
- Automatically cleaned up when program exits
- Support for multiple programs on same hook

# BTF: The Type System Foundation

**BTF Format**

- Binary format encoding C type information
- Compact representation with string deduplication
- Enables CO-RE relocations and type verification

**Kernel BTF Registry**

- vmlinux BTF: Core kernel types
- Module BTF: Per-module types
- User BTF: Object-specific types
- Type ID remapping for compatibility

# Map FD Relocation

- Compiler generates placeholders in map access instructions
- Libbpf parses `.maps` definition and creates maps
- Libbpf stores FDs in `obj->fd_array[]`
- Program instructions are patched with FD values
- The verifier will translate FD to map pointer

# Relocation Engine

## CO-RE Relocation Types:

```
enum bpf_core_relo_kind {
    BPF_RELO_FIELD_BYTE_OFFSET,    // struct field access
    BPF_RELO_FIELD_BYTE_SIZE,      // sizeof operations
    BPF_RELO_TYPE_ID_TARGET,       // kernel type matching
    BPF_RELO_ENUMVAL_VALUE,        // enum value resolution
};
```

## Internal Relocation Types:

```
enum reloc_type {
    RELO_LD64,         // Map loading (FD → instruction)
    RELO_CALL,         // Function calls
    RELO_DATA,         // Global data access
    RELO_EXTERN_LD64, // External symbols
    RELO_CORE,         // CO-RE relocations
};
```

# Example: Map Relocation

## Map Reference in Program:

```
// BPF program referencing a map
struct bpf_map_def SEC("maps") my_map = { /* ... */ };

SEC("kprobe/sys_open")
int trace_open(void *ctx) {
    bpf_map_lookup_elem(&my_map, &key); // This generates RELO_LD64 relocation
}
```

## Relocation Processing:

```
case RELO_LD64:  // Map FD relocation
    map_fd = obj->maps[relo->map_idx].fd;
    // Patch instruction with actual FD
    insn->src_reg = BPF_PSEUDO_MAP_FD;
    insn->imm = map_fd;
    break;
```

# CO-RE (Compile Once, Run Everywhere)

**The Problem:**

- Kernel structures change across versions
- Field offsets differ between kernels
- Traditional eBPF programs break on kernel updates

# CO-RE Solution

```
// In BPF program - symbolic access
struct task_struct *task = ...;
int pid = BPF_CORE_READ(task, pid);  // Field offset unknown at compile time

// At load time - resolved via BTF by the kernel
case BPF_RELO_FIELD_BYTE_OFFSET:
    offset = btf_member_bit_offset(target_btf, member_idx);
    bpf_core_patch_insn(prog_name, insn, insn_idx, relo, relo_idx, &targ_res);
```

**Benefits:**

- Same binary runs on multiple kernel versions
- Type-safe field access with BTF validation
- Automatic adaptation to kernel structure changes

# External Symbol Resolution

**Kernel Feature Adaptation:**

- Programs adapt behavior based on kernel capabilities
- Feature detection without runtime checks
- Compile-time optimization with runtime flexibility

# Resolution Process

## External Symbol Types:

- `EXT_KCFG` - Kernel config symbols (CONFIG_*)
- `EXT_KSYM` - Kernel symbols (/proc/kallsyms)
- `EXT_KFUNC` - Kernel function addresses
- `EXT_DATASEC` - Data section references

## Resolution Process:

```
// Example: kernel config resolution
extern bool CONFIG_BPF_SYSCALL __kconfig; // in your eBPF code

// libbpf: read /proc/config.gz or /boot/config-*
bpf_object__read_kconfig_file(obj, kcfg_data);  // Sets CONFIG_BPF_SYSCALL = 1 or 0
```

# Kernel Verifier Interaction

### BPF_PROG_LOAD Attributes:

```
union bpf_attr attr = {
    .prog_type = prog->type,          // XDP, kprobe, etc
    .insn_cnt = prog->insns_cnt,      // Instruction count
    .insns = ptr_to_u64(prog->insns), // Program bytecode
    .log_size = log_buf_size,         // Log buffer size
    .log_buf = ptr_to_u64(log_buf),   // Verifier output
    .prog_btf_fd = prog->btf_fd,      // Type information
};
```

### Syscall and Response:

- `fd = syscall(__NR_bpf, BPF_PROG_LOAD, &attr);`
- Success: Returns program FD
- Failure: Returns -1, verifier log has details

# Verifier Log Processing

**BPF Verifier Visualizer (bpfvv):**

- Interactive tool for debugging verification failures
- Parses verifier logs and visualizes program states
- Acts as primitive debugger UI for BPF programs
- Load text files containing verifier output for analysis
- Available at: https://libbpf.github.io/bpfvv/

# Program Signing Challenges

**Current State:**

- No built-in signing mechanism in current libbpf and kernel

**Key Challenges:**

- Post-relocation signing complexity
- Kernel version compatibility
- Dynamic symbol resolution

**Proposed Solutions:**

- 2-phase Signing: https://github.com/congwang/ebpf-2-phase-signing
- Hornet LSM: light-skeleton-based eBPF signature verification

# Key Takeaways

1. **Layered Design**: libbpf provides sophisticated abstraction over kernel complexity
2. **Type Safety**: BTF enables portable, type-safe program development
3. **Smart Relocation**: CO-RE technology enables true portability
4. **Robust Verification**: Multi-stage validation ensures program safety

**Bottom Line**: eBPF program loading is a carefully orchestrated dance between userspace tooling and kernel verification

# Thank You

**Key Resources:**

- libbpf source: `github.com/libbpf/libbpf`
- eBPF documentation: `ebpf.io`
- Kernel BPF documentation: `kernel.org/doc/html/latest/bpf/`
- Kernel mailing list: `bpf@vger.kernel.org`