facebook

Bringing BPF dev experience to the next level

Andrii Nakryiko

Developing BPF application (today)

Development server

bpf.c

```
#include <linux/bpf.h>
#include <linux/filter.h>
int prog(struct __sk_buff* skb)
{
    if (skb->len < X) {
        return 1;
    }
    ...
}
```

embed

ControlApp.cpp

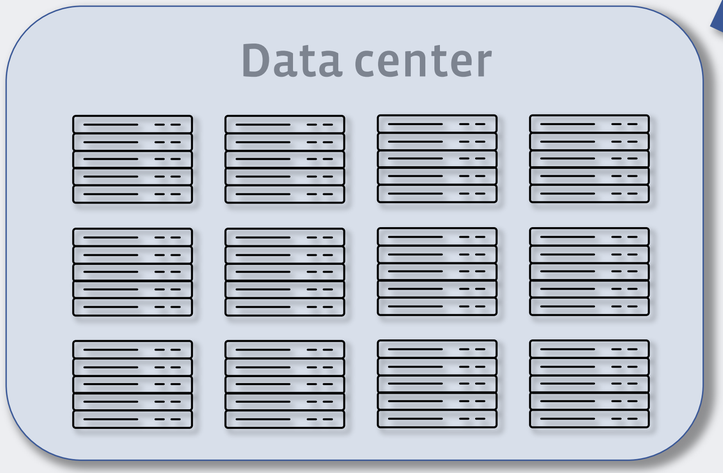
```
#include <bcc/BPF.h>

std::string BPF_PROGRAM =
#include ”path/to/bpf.c”

namespace facebook {
    . . .
}
```

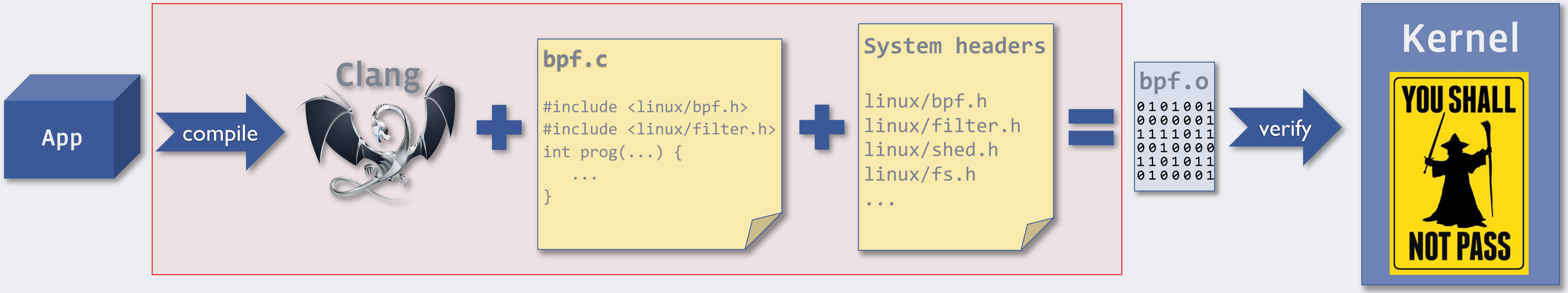
compile

deploy



Developing BPF application (today)

Production server



“On the fly” BPF compilation

Problems

- Dependency on system **kernel headers**
- LLVM/Clang dependency is **big and heavy**
- Compilation errors **captured in runtime only**

BPF CO-RE

(Compile Once – Run Everywhere)

Solution

- ~~Dependency on system **kernel headers**~~
- **Generate** vmlinux.h from kernel's **BTF**
- ~~LLVM/Clang dependency is **big and heavy**~~
- **Pre-compile** BPF program w/ **BTF** relocations
- ~~Compilation errors **captured in runtime only**~~
- **Pre-validate** relocations against multiple kernel **BTFs**

Kernel BTF

Freedom from system headers dependency

- CONFIG_DEBUG_INFO_BTF=y (needs pahole >= v1.13)
- Compact, always in sync, at fixed location:

```
$ ls -lah /sys/kernel/btf/vmlinux
-r--r--r-- 1 root root 2.2M Aug 17 22:02
```

- **All types** w/ lossless BTF to C conversion:

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

- No `#defines` – prefer enums for constants and flags

DONE: Field offset relocation

```
#include ”vmlinux.h”
#include <bpf_core.h>




int on_event(void *ctx) {
    struct task_struct *task;
    u64 read_bytes;

    task = (void *)bpf_get_current_task();

      read_bytes =
          BPF_CORE_READ(task, ioac.read_bytes);
    /* __builtin_preserve_access_index() */
    return 0;
}
```

```
0: (85) call bpf_get_current_task
1: (07) r0 += 1952
2: (bf) r1 = r10
3: (07) r1 += -8
4: (b7) r2 = 8
5: (bf) r3 = r0
6: (85) call bpf_probe_read
7: (b7) r0 = 0
8: (95) exit
```

```
Field reloc:
  - insn: #1
  - type: struct task_struct
  - accessor: 30:4
```

TODO: Conditional relocation

```
#include “vmlinux.h”
#include <bpf_core.h>

/* relies on /proc/config.gz */
extern bool CONFIG_IO_TASK_ACCOUNTING;

int on_event(void *ctx) {
    struct task_struct *task;
    u64 read_bytes;

    task = (void *)bpf_get_current_task();
    if (CONFIG_IO_TASK_ACCOUNTING) {
      read_bytes =
          BPF_CORE_READ(task, ioac.read_bytes);
    }
    return 0;
}
```

```
 0: (85) call bpf_get_current_task
 1: (b7) r1 = XXX
 2: (15) if r1 == 0x0 goto pc+6
 3: (07) r0 += 1952
 4: (bf) r1 = r10
 5: (07) r1 += -8
 6: (b7) r2 = 8
 7: (bf) r3 = r0
 8: (85) call bpf_probe_read
 9: (b7) r0 = 0
10: (95) exit

Extern reloc:
  - insn: #1
  - name: CONFIG_TASK_IO_ACCOUNTING
  - type: bool

Field reloc:
  - insn: #3
  - type: struct task_struct
  - accessor: 30:4
```

Discussion: conditional relocation impl

- Clang support (experimental) by Yonghong
- Emits special kind of BTF relocation for

```
extern int LINUX_VERSION SEC(".BPF.patchable_externs");
```

- Rewrite instructions into immediate constant loads
- Works only on up to **8 byte variables**

Discussion: conditional relocation impl

- Researching more generic support:

```
struct my_struct { int a, b; };

extern struct my_struct my_struct;

int on_event(void* ctx) {
    if (my_struct.b > 10) {
        ...
    }
    return 0;
}
```

```
1: r1 = XXX
2: r1 = *(u32 *)(r1 + 4)
3: (15) if r1 > 0xA goto pc+7
...
10: (95) exit
```

- Create new internal map for externs, make it **read-only**
- Teach verifier to track constants from this map

Beyond CO-RE

Making typical uses simple and easy

- BTF-defined maps
- Control app usability
- Global data usability

BTF-defined maps

BTF-defined maps

- Easily extensible
- Purely type declaration-based
- Capture key/value BTF type automatically
- Enable advanced scenarios w/ great usability

BTF-defined maps

Simple maps

BEFORE

```
struct bpf_map_def my_map SEC(”maps”) = {
    .type = BPF_MAP_TYPE_HASH,
    .max_entries = 1,
    .key_size = sizeof(int),
    .value_size = sizeof(struct my_value),
};

BPF_ANNOTATE_KV_PAIR(my_map,
    int, struct my_value);
```

NOW

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH),
    __uint(max_entries, 1),
    __type(key, int),
    __type(value, struct my_value),
} my_map SEC(“.maps”);
```

BTF-defined maps

How to define map-in-map?

IPROUTE2 WAY

```
#define MAP_INNER_ID    42

struct bpf_elf_map __section_maps map_inner = {
        .type           = BPF_MAP_TYPE_ARRAY,
        .size_key       = sizeof(uint32_t),
        .size_value     = sizeof(uint32_t),
        .id             = MAP_INNER_ID,
        .inner_idx      = 0,
        .max_elem       = 1,
};

struct bpf_elf_map __section_maps map_outer = {
        .type           = BPF_MAP_TYPE_ARRAY_OF_MAPS,
        .size_key       = sizeof(uint32_t),
        .size_value     = sizeof(uint32_t),
        .inner_id       = MAP_INNER_ID,
        .max_elem       = 1,
};
```

BTF-DEFINED WAY?

```
struct inner_map_template {
        __uint(type, BPF_MAP_TYPE_ARRAY),
        __uint(max_entries, 1),
        __type(key, uint32_t),
        __type(value, uint32_t),
} map_inner SEC(".maps");

struct {
        __uint(type, BPF_MAP_TYPE_ARRAY_OF_MAPS),
        __uint(max_entries, 1),
        __type(key, uint32_t),
        __array(values, struct inner_map_template),
} map_outer SEC(".maps") = {
        .values = { &map_inner },
};
```

BTF-defined maps

How to define map-in-map?

IPROUTE2 WAY: LOTS OF DUPLICATION

```
#define MAP_INNER_ID    42
struct bpf_elf_map __section_maps map_inner1 = {
        .type           = BPF_MAP_TYPE_ARRAY,
        .size_key       = sizeof(uint32_t),
        .size_value     = sizeof(uint32_t),
        .id             = MAP_INNER_ID,
        .inner_idx      = 0,
        .max_elem       = 1,
};
struct bpf_elf_map __section_maps map_inner2 = {
        .type           = BPF_MAP_TYPE_ARRAY,
        .size_key       = sizeof(uint32_t),
        .size_value     = sizeof(uint32_t),
        .id             = MAP_INNER_ID,
        .inner_idx      = 1,
        .max_elem       = 1,
};
struct bpf_elf_map __section_maps map_outer = {
        .type           = BPF_MAP_TYPE_ARRAY_OF_MAPS,
        .size_key       = sizeof(uint32_t),
        .size_value     = sizeof(uint32_t),
        .inner_id       = MAP_INNER_ID,
        .max_elem       = 2,
};
```

BTF-DEFINED WAY?

```
struct inner_map_template {
        __uint(type, BPF_MAP_TYPE_ARRAY),
        __uint(max_entries, 1),
        __type(key, uint32_t),
        __type(value, uint32_t),
};

struct inner_map_template map_inner1 SEC(".maps");
struct inner_map_template map_inner2 SEC(".maps");

struct {
        __uint(type, BPF_MAP_TYPE_ARRAY_OF_MAPS),
        __uint(max_entries, 2),
        __type(key, uint32_t),
        __array(values, struct inner_map_template),
} map_outer SEC(".maps") = {
        .values = { &map_inner1, &map_inner2 },
};
```

BTF-defined maps

Tail call array (PROG_ARRAY)

IPROUTE2: SPECIAL CONVENTIONS

```
struct bpf_elf_map __section_maps jmp_tc = {
     .type           = BPF_MAP_TYPE_PROG_ARRAY,
     .id             = 123,
     .size_key       = sizeof(uint32_t),
     .size_value     = sizeof(uint32_t),
     .max_elem       = 2,
};

SEC("123/0")
int cls_case1(struct __sk_buff *skb) { ... }

SEC("123/1")
int cls_case2(struct __sk_buff *skb) { ... }
```

BTF-DEFINED WAY: SAME INITIALIZATION

```
typedef int (*tail_call_fn)(struct __sk_buff *);

SEC("custom_name_1")
int cls_case1(struct __sk_buff *skb) { ... }

SEC("custom_name_2")
int cls_case2(struct __sk_buff *skb) { ... }

struct {
        __uint(type, BPF_MAP_TYPE_PROG_ARRAY),
        __uint(max_entries, 2),
       __array(values, tail_call_fn),
} jmp_tc SEC(".maps") = {
        .values = { &cls_case1, &cls_case2 },
};
```

Control app usability

Control app usability

Runqslower example (from BCC tools)

BPF (kernel) side:

```
struct { ... } start SEC(".maps");

struct { ... } events SEC(".maps");

SEC("raw_tracepoint/sched_wakeup")
int handle__sched_wakeup(struct bpf_raw_tracepoint_args *ctx) { ... }

SEC("raw_tracepoint/sched_switch")
int handle__sched_switch(struct bpf_raw_tracepoint_args *ctx) { ... }
```

Control app usability

Runqslower control app boilerplate

Lots of boilerplate!

1. Embed BPF object file as a variable (using **.incbin** assembler directive):
   ```
   BPF_EMBED_OBJ(runqslower_bpf, "runqslower.bpf.o");
   ```
2. **bpf_object__open_buffer**() + handle errors
3. **bpf_object__load**() + handle errors
4. Lookup PERF_EVENT_ARRAY BPF map to set up perf buffer (**bpf_object__find_map_by_name**())
5. Global data initialization is a separate topic...
6. For each program:
   1. **bpf_object__find_program_by_title**()
   2. attach and remember **bpf_link**
7. Clean up links, close object

Control app usability

Solution: pre-generate BPF object-specific template

```
struct runqslower_bpf {
    struct bpf_obj_template *template;

    struct bpf_object *obj;
    struct {
        struct bpf_map *start;
        struct bpf_map *events;
    } map;
    struct {
        struct bpf_program *sched_wakeup;
        struct bpf_program *sched_switch;
    } prog;
    struct {
        struct bpf_link *sched_wakeup;
        struct bpf_link *sched_switch;
    } link;
};
```

Control app usability

BPF object template APIs?

1. Bpftool can generate “idiomatic” code for multiple languages (not just C)
2. First, **bpf_obj_template__load** (runqslower_bpf.template);
3. Perform set up, access maps and programs directly, e.g.:
   - runqslower_bpf.**map.events**
   - runqslower_bpf.**prog.sched_new**
4. **bpf_obj_template__attach** (runqslower_bpf.template);
5. When done, **bpf_obj_template__destroy** (runqslower_bpf.template).

Control app usability

Flow

- Makefile: generate vmlinux.h (included by BPF program)
- Makefile: build bpf.o
- Makefile: generate bpf object template
- Control app: use generated BPF object template:
  - Load and attach BPF object with simple API
  - Access maps, progs, variables, etc, **directly as struct fields**
- **It’s almost like having custom-tailored libbpf ☺**

Global data usability

Global data usability (today)

BPF (kernel) side

```
static volatile struct {
    pid_t pid;
    __u64 min_delay_us;
} opts;

int probe(struct pt_regs *ctx)
{
    pid_t cur_pid = bpf_get_current_pid_tgid() >> 32;

    if (opts.pid != cur_pid)
        return 0;
    ...
}
```

Global data usability (today)

Control app (user) side

```
int zero = 0;
struct {
    pid_t pid;
    __u64 min_delay_us;
} opts = { XXX, YYY };

/* no easy way to update only pid or min_delay_us */

struct bpf_map* opts_map = bpf_object__find_map_by_name(obj, "runqslow.bss");
if (bpf_map_update_elem(bpf_map__fd(opts_map), &zero, &opts, 0)) {
    ...
}
```

Global data usability

- Global data is awesome, but hard to use from user space
- All or nothing read/update
- With multiple variables, need to know exact offsets
- Need to share struct definition (much simpler with BPF CO-RE now)
- Solution: rely on **BTF** and **code generation**?
  - We know name, type, offset, size of each variable
  - Generate types/accessor fields in BPF object template
  - **Problem: need kernel to support partial map reads**

Discussion: map partial read/update

- Support mmap() for BPF_MAP_TYPE_ARRAY:
  - Lowest overhead
  - Best usability
  - **Only works for single-element map?** (ok for global data)
- Generic partial map value read/update:
  - BPF_MAP_LOOKUP_ELEM/ BPF_MAP_UPDATE_ELEM get optional `offset` and `size` attributes
  - Syscall overhead, need libbpf API wrappers for access
  - **Works for (almost) any map?**

Global data usability

Control app (user) side

```
pid_t old_pid, new_pid = XXX;

struct bpf_var *pid_var = bpf_object__find_var_by_name(obj, “pid”);
bpf_var__read(pid_var, &old_pid);
bpf_var__update(pid_var, &new_pid);

/* With bpf_obj_template */
bpf_var__read(runqslower_bpf.var.pid, &old_pid);
bpf_var__update(runqslower_bpf.var.pid, &new_pid);
```

facebook