BPF 系统探测及其原理

毛奕夫

目录

§1 简介

BPF 让我们可以在内核中安全地执行某些代码。现在人们常说的 BPF 往往指 eBPF(extended BPF)，而不是 1992 年首次面世的仅仅用于数据包过滤的 cBPF(classic BPF)。自 2013 年 cBPF 被重写升级为 eBPF(后文简称 BPF) 后，BPF 能做的除了数据包的过滤之外，还可以和内核的其它组件配合，进行流量控制 (tc)、高效处理数据包 (XDP)、资源访问控制 (cgroup) 以及系统探测 (krpobe，uprobe，tracepoint 和 perf event)。本文主要关心 BPF 的系统探测方面。

BPF 程序有自己的一套类似 RISC 的指令集。由 C 语言编写的 BPF 程序通过 LLVM/clang 编译成 BPF 对象文件，其中包含了 BPF 指令及其它信息。BPF 指令可以通过后面讲到的 libbpf 或 BCC 从 BPF 对象文件中提取出来并加载进 (注入) 内核。BPF 指令的安全性验证通过后，会通过内核中的解释器解释执行，或者通过内核中的 JIT 编译器编译成本地机器指令后再执行。

BPF 程序的加载、映射 (见下文) 的操作通过 `bpf()` 系统调用完成，将加载后的 BPF 程序附加到某个被测函数又通过 `perf_event_open()` 和 `ioctl()` 两个系统调用完成。直接使用这些系统调用优点是

灵活——缺点更显而易见，那就是麻烦。为了避免重复构建这些系统调用需要的参数，我们可以用 BCC 或 libbpf。这两套函数库分别把上述系统调用作了封装，其中 BCC 提供的 API 相当于是 libbpf 的超集。BCC 内部调用了 libbpf 的某些函数，同时提供了一些额外的特性。

本文主要对比了 libbpf、BCC 以及 BCC 的 Python 绑定使用上的一些相同点和不同点，简述了 ring buffer 的使用，并简单分析了三者某些功能的原理。

为使代码简介，本文中的部分代码没有包含错误处理。

§2 BPF 应用的要素

这里列举一些编写 BPF 应用的过程中反复出现的概念和注意的点。

- 适当的头文件。由于 BPF 应用的编写一般分为两部分，一部分是要加载进内核的 BPF 程序本身，运行在内核空间，一部分是用户空间的 (控制) 程序，负责调用 BCC 或 libbpf 的 API 进行 BPF 程序、映射的加载以及信息的获取。两个部分的头文件不同，BPF 程序部分主要使用包含 BPF 帮助函数、被测函数上下文处理、内核类型定义的头文件；用户空间程序主要使用包括操作 BPF 程序、映射以及其它应用逻辑需要的头文件。

- BPF 映射 (map)。BPF 程序通常会被附加 (attach) 到某个被测函数并搜集相关的信息，这些信息再又用户程序获取。映射就是两种程序交换信息的桥梁。映射有多种类型，功能各异，详见 `linux/bpf.h` 中的 `bpf_map_type` 枚举体。

- BPF 程序。一个 BPF 程序在源代码中对应一个 C 语言函数，一份 C 语言源代码可以定义多个 BPF 程序，对应多个 C 函数。在下文中这样的 C 函数被称为 BPF 函数。BPF 程序也有多种类型，详见 `linux/bpf.h` 中的 `bpf_prog_type` 枚举体。BPF 程序为事件驱动，也就是说只有当内核中发生相应的事件时 (如某个内核函数被调用)，相应的 BPF 程序才会被执行。此时该 BPF 程序可以访问这个内核函数的上下文。

- 被测函数的上下文。被测的函数都有一个上下文，从中可以获取这个函数的各个参数或返回值。对 kprobe 来说这个上下文放在 `struct pt_regs *ctx` 里面，对于 tracepoint 而言我们需要根据特定的函数自己构造一个结构体来存放上下文。

- BPF 帮助函数 (helper functions)。为了系统的稳定，BPF 程序的功能是受限制的。在内核空间运行的 BPF 程序使用不了 C 标准库，但 BPF 虚拟机提供了一套函数供 BPF 程序使用，这套函数称为 BPF 帮助函数。操作映射、获取进程相关信息、从指定的地址读取数据等操作都通过帮助函数完成。详见 *bpf-helpers(7)*。

- 特性 (attibute)。如果不是手写 BPF 汇编，而是以 C 语言函数的形式编写 BPF 程序，那 libbpf 或者 BCC 需要某种机制来获取编译好的 BPF 对象文件中的 BPF 程序 (前面说过，一份 C 语言源代码可以定义多个 BPF 程序)。获取 BPF 程序的机制依赖 ELF 格式的节区 (section)。一个 ELF 文件包含不同的节区，每个节区都有一个名字。通常情况下，程序在编译后会默认生成几个节区，如包含机器指令的 `.text` 和数据的 `.data`。我们通过特性告诉编译器某个函数或某个变量应该被放在哪一个节区，从而方便对 ELF 的解析。

§3 两个 BPF 程序

这一节我们分别使用 kprobe 和 tracepoint 两种技术来探测内核的函数，以此展示通过 BPF 探测系统的方法。

3.1 kprobe/vfs_read

Linux 的虚拟文件系统 (VFS) 抽象了下层不同的文件系统实现，向上提供了统一的文件操作接口。`vfs_read()` 负责文件的读取，通过探测这个函数我们可以了解系统的文件读取情况，进而发现可能的 IO 瓶颈。`vfs_read()` 没有固定的 tracepoint，所以我们需用 kprobe 探测此函数。

现在假设我们需要记录每一个进程读取文件的次数，并把进程名、PID 和该进程调用 `vfs_read()` 的次数返回给用户空间。这里我们使用哈希映射 (`BPF_MAP_TYPE_HASH`) 来存储和传递数据，其键是一个包含进程名和 PID 的结构体，其值为一个无符号整数，用来记录 `vfs_read()` 的调用次数。之所以将一个包含进程名和 PID 的结构体作为该哈希映射的键是为了方便确定每一次触发 `vfs_read()` 被调用这个事件的程序是否已经在映射中，如果在，则将其值加一，否则将和这一次 `vfs_read()` 调用相关的进程名和 PID 作为键存入映射，并将相关联的值设为一。于是上述结构体和映射的定义如下：

```c
#inlcude <bpf/bpf_helpers.h>

struct map_key {
    char comm[16];
    u32 pid;
};

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(struct map_key),
    .value_size = sizeof(u32),
    .max_entries = 1024
};
```

这里我们声明映射的方式是创建一个 `struct bpf_map_def` 结构，其定义在头文件 `bpf/bpf_helpers.h` 中。另外，这个头文件也包含了 BPF 帮助函数的声明以及创建结构体用到的 `SEC(NAME)` 宏。这个宏展开实际上利用了编译器的特性机制：

```c
__attribute__((section(NAME), used))
```

意思是告诉编译器，将被上述代码修饰的变量或函数编译后放进名为 `NAME` 的 ELF 节区，这样做可以方便 libbpf 和 BCC 从编译好的对象文件中提取对应的数据。

有了前面的逻辑，下面给出 BPF 程序的代码：

```c
SEC("kprobe/vfs_read")
int vfs_read_probe(struct pt_regs *ctx)
{
    struct map_key k;
    u32 init = 1;
    u32 *res;

    k.pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&k.comm, sizeof(k.comm));

    res = bpf_map_lookup_elem(&my_map, &k);
```

```
    if (res) {
        (*res)++;
    } else {
        bpf_map_update_elem(&my_map, &k, &init, BPF_ANY);
    }

    return 0;
}
```

ctx 表示被测函数 (这里的 vfs_read()) 的上下文——实际上就是当被测函数运行时处理器寄存器的值，因为这些值包含了被测函数的参数的值或指向这些参数的指针。如果用的 kretprobe，则特定的寄存器包含了被测函数的返回值或其指针。struct pt_regs 的成员变量依具体的处理器架构的不同而不同。在 x86_64 的机器上，其定义如下：

```
struct pt_regs {
    long unsigned int r15;
    long unsigned int r14;
    long unsigned int r13;
    long unsigned int r12;
    long unsigned int bp;
    long unsigned int bx;
    long unsigned int r11;
    long unsigned int r10;
    long unsigned int r9;
    long unsigned int r8;
    long unsigned int ax;
    long unsigned int cx;
    long unsigned int dx;
    long unsigned int si;
    long unsigned int di;
    long unsigned int orig_ax;
    long unsigned int ip;
    long unsigned int cs;
    long unsigned int flags;
    long unsigned int sp;
    long unsigned int ss;
};
```

x86_64 的 ABI 规定了进行函数调用时第一个参数应该放在 di 寄存器，第二个放在 si……所以我们在上面的 vfs_read_probe() 函数中可以对 ctx->di 做相应的类型转换来获取 vfs_read() 的第一个参数，其它的参数同理。事实上，libbpf 提供了方便的宏 (bpf/bpf_tracing.h) 来提取某个寄存器的值，如第一个寄存器用 PT_REGS_PARM1() 访问，第二个用 PT_REGS_PARM2() 访问，如此种种。这里我们只关心 vfs_read() 的的调用次数，所以不会用到这类宏。

struct pt_regs 的定义并不在一般文章所说的 linux/ptrace.h 中。这里我们建议手动生成一个 vmlinux.h 文件[1]，再将其放在 kern.c 的同级目录：

[1]这里用到了 BTF(BTF Type Format)。BTF 和 DWARF 类似，都存储了程序的调试信息。例如，C 语言的结构、枚举、联合的定义都会存储在调试信息中方便调试。BTF 和 DWARF 相比更加简洁紧凑。启用了 BTF 的内核 (编译选项：CONFIG_DEBUG_INFO_BTF=y) 会在/sys/kernel/btf/vmlinux 暴露自己的类型信息，通过 bpftool 可以将其转化成一个 C 语言头文件供 BPF 程序使用。

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

这样生成的头文件包含了当前机器内核的几乎所有类型定义，自然也包括了 `struct pt_regs`。

之后以 `bpf_` 开头的均为 BPF 帮助函数，详见 *bpf-helpers(7)*。

总的代码如下：

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

struct map_key {
    char comm[16];
    u32 pid;
};

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(struct map_key),
    .value_size = sizeof(u32),
    .max_entries = 1024
};

SEC("kprobe/vfs_read")
int vfs_read_probe(struct pt_regs *ctx)
{
    struct map_key k;
    u32 init = 1;
    u32 *res;

    k.pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&k.comm, sizeof(k.comm));

    res = bpf_map_lookup_elem(&my_map, &k);
    if(res) {
        (*res)++;
    } else {
        bpf_map_update_elem(&my_map, &k, &init, BPF_ANY);
    }

    return 0;
}

char _license[] SEC("license") = "GPL";  /* 告诉内核我们的 BPF 程序协议 */
```

可以将其保存为 `kern.c`，编译：

```
$ clang -O2 -target bpf -c kern.c -o kern.o
```

3.2 tracepoint/syscalls/sys_enter_execve

`execve()` 系统调用用于将当前进程替换成一个新的进程，通过探测此函数可以了解系统进程的创建

情况[2]。

查阅 *execve(2)* 手册可知此函数的签名：

```c
#include <unistd.h>

int execve(const char *path, char *const argv[], char *const envp[]);
```

该函数的三个参数分别表示被执行程序的路径、传递给它的命令行参数和环境变量。假设我们想获取调用这个系统调用的进程名、PID 以及被调用进程的路径，则 BPF 函数如下：

```c
SEC("tracepoint/syscalls/sys_enter_execve")
int prog(struct execve_args *ctx)
{
    struct data d;
    d.pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&d.comm, sizeof(d.comm));
    bpf_probe_read_user_str(&d.filename, sizeof(d.filename), ctx->filename);

    u64 key = bpf_ktime_get_coarse_ns();
    bpf_map_update_elem(&my_map, &key, &d, BPF_ANY);

    return 0;
}
```

函数的参数 `ctx` 是一个自定义结构体。探测拥有 tracepoint 的函数的 BPF 函数所使用的参数和 kprobe 不同，我们需要自己确定被探测函数的参数的类型，从而确定 BPF 函数的参数。相关的信息通过 tracefs/debugfs 文件系统获取[3]。这个文件系统默认挂载在`/sys/kernel/debug`，如果你的系统没有，请运行 `mount` 命令检查。该文件系统下的 `tracing` 目录包含了和系统追踪相关的文件和目录。现在默认当前工作目录是`/sys/kernel/debug/tracing`，其中 `available_events` 列举了所有拥有 tracepoint 的内核或内核模块函数，`events` 目录包含了这些函数更详细的信息。和系统调用相关的信息均放在 `events/syscalls/`目录下。由于我们可以分别探测每一个系统调用的进入和返回，所以每个系统调用在该目录下对应两个目录。以 `execve()` 为例，在 `events/syscalls/`我们有 `sys_enter_execve` 和 `sys_exit_execve` 两个目录。这里我们只看前者：

```
$ ls events/syscalls/sys_enter_execve

enable  filter  format  hist  id  trigger
```

列出的四个文件中的 `format` 是我们需要的：

```
$ cat events/syscalls/sys_enter_execve/format

name: sys_enter_execve
ID: 718
format:
```

[2]类似创建进程的函数还有 `fork()`，`vfork()` 以及 C 库函数 `system()` 等。

[3]debugfs 用于向用户空间展示内核信息；tracefs 基于 ftrace，用于多种形式的内核追踪。两个虚拟文件系统的默认挂载点分别为`/sys/kernel/tracing` 和`/sys/kernel/debug`。小于 4.1 版本的内核中 tracefs 的内容都在 debugfs 的 `tracing` 目录下，即`/sys/kernel/debug/tracing`；4.1 之后的内核有了一个独立的 tracefs 文件系统，而 debugfs 中的 `tracing` 目录依然存在，两者内容完全一致。

```
field:unsigned short common_type;        offset:0;        size:2;        signed:0;
field:unsigned char common_flags;        offset:2;        size:1;        signed:0;
field:unsigned char common_preempt_count;        offset:3;        size:1;        signed:0;
field:int common_pid;        offset:4;        size:4;        signed:1;

field:int __syscall_nr;        offset:8;        size:4;        signed:1;
field:const char __attribute__((user)) * filename;        offset:16;        size:8;...
field:const char __attribute__((user)) *const __attribute__((user)) * argv;        offset:24...
field:const char __attribute__((user)) *const __attribute__((user)) * envp;        offset:32...

print fmt: "filename: 0x%08lx, argv: 0x%08lx, envp: 0x%08lx", ((unsigned long)(REC->filename))...
```

可以看出，format 中包含了该 tracepoint 的名称、ID 和格式等信息。以 field 开头的行列举了和 execve() 相关的字段，其中以 common_ 开头的字段属于不同的 tracepoint 共有的，这里我们不管。后面的字段和 execve() 系统调用有关，其中 __syscall_nr 代表该系统调用的编号，在不同的 CPU 架构上相同的系统调用可能有不同的编号。后面的三个字段即对应 execve() 的三个参数。

每一个字段的 offset、size 和 signed 和其数据类型相对应，分别代表该字段 (相对于第一个字段) 的偏移、大小和是否是有符号的。前两者以字节为单位，后者为 1 表示有符号，为 0 则表示无符号。

由 __syscall_nr 的偏移可知，前四个 common_ 总共占 8 个字节，即 64 位。虽然我们不需要这四个值，但系统要求作为 BPF 函数参数的结构体的大小和所有字段的大小保持一致，所以我们使用一个 u64 类型的变量将它们覆盖起来。于是 struct execve_args 的定义如下：

```c
struct execve_args {
    u64 unused;
    int sys_nr;
    u64 *filename;
    u64 *argv;
    u64 *envp;
};
```

结构中成员变量的类型和 format 中列出的保持一致即可。

BPF 函数中出现的 struct data 盛放我们搜集的调用 execve() 的进程名、PID 以及被调用进程的路径，其定义如下：

```c
struct data {
    char comm[16];
    unsigned int pid;
    char filename[32];
};
```

之后我们定义一个哈希映射向用户空间传递信息：

```c
struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(u64),
    .value_size = sizeof(struct data),
    .max_entries = 1024
};
```

总的代码如下：

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

struct data {
    char comm[16];
    unsigned int pid;
    char filename[32];
};

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(u64),
    .value_size = sizeof(struct data),
    .max_entries = 1024
};

struct execve_args {
    u64 unused;
    int sys_nr;
    u64 *filename;
    u64 *argv;
    u64 *envp;
};

SEC("tracepoint/syscalls/sys_enter_execve")
int prog(struct execve_args *ctx)
{
    struct data d;
    d.pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&d.comm, sizeof(d.comm));
    bpf_probe_read_user_str(&d.filename, sizeof(d.filename), ctx->filename);

    u64 key = bpf_ktime_get_coarse_ns();
    bpf_map_update_elem(&my_map, &key, &d, BPF_ANY);

    return 0;
}

char _license[] SEC("license") = "GPL";
```

编译同上：

```
$ clang -O2 -target bpf -c kern.c -o kern.o
```

§4 libbpf

libbpf 是编写 BPF 应用的官方函数库，以下是 libbpf 中常用的 API：

```c
#include <bpf/libbpf.h>

// 加载 BPF 对象文件
struct bpf_object *bpf_object__open(const char *path);

// 关闭 BPF 对象
void bpf_object__close(struct bpf_object *object);

// 通过函数名获取 BPF 程序
struct bpf_program *
bpf_object__find_program_by_name(const struct bpf_object *obj,
                                 const char *name);

// 通过节区名获取 BPF 程序
struct bpf_program *
bpf_object__find_program_by_title(const struct bpf_object *obj,
                                  const char *title);

// 加载 BPF 程序
int bpf_object__load(struct bpf_object *obj);

// 附加探针
struct bpf_link * bpf_program__attach(struct bpf_program *prog);

struct bpf_link *
bpf_program__attach_kprobe(struct bpf_program *prog, bool retprobe,
                           const char *func_name);

struct bpf_link *
bpf_program__attach_uprobe(struct bpf_program *prog, bool retprobe,
                           pid_t pid, const char *binary_path,
                           size_t func_offset);

struct bpf_link *
bpf_program__attach_tracepoint(struct bpf_program *prog,
                               const char *tp_category,
                               const char *tp_name);

// 关闭探针
int bpf_link__destroy(struct bpf_link *link);

// 通过名字获取映射的描述符
bpf_object__find_map_fd_by_name(const struct bpf_object *obj, const char *name);

// 映射操作
#include <bpf/bpf.h>

int bpf_map_update_elem(int fd, const void *key, const void *value, __u64 flags);
int bpf_map_lookup_elem(int fd, const void *key, void *value);
int bpf_map_lookup_elem_flags(int fd, const void *key, void *value, __u64 flags);
int bpf_map_lookup_and_delete_elem(int fd, const void *key, void *value);
```

```
52 int bpf_map_delete_elem(int fd, const void *key);
53 int bpf_map_get_next_key(int fd, const void *key, void *next_key);
```

BPF 应用的编写往往有固定的模式。假设我们已经将 BPF 代码编译成 `kern.o`，首先打开这个文件，获取一个 `struct bpf_object` 指针：

```
1 struct bpf_object *obj;
2 obj = bpf_object__open("kern.o");
```

之后从中找到 BPF 程序（以前面的 `kprobe/vfs_read` 为例）：

```
1 struct bpf_program *prog;
2 // 按节区名
3 prog = bpf_object__find_program_by_title(obj, "kprobe/vfs_read");
4 // 按函数名
5 prog = bpf_object__find_program_by_name(obj, "vfs_read_probe");
```

附加探针：

```
1 struct bpf_link *link;
2 link = bpf_program__attach(prog);
```

如果 BPF 代码中通过 `SEC()` 指定了节区名，且格式正确，如`"kprobe/vfs_read"` 或`"tracepoint/syscalls/sys_enter_execve"`，则可以直接用 `bpf_program__attach()` 智能附加探针，libbpf 内部会自动判断 BPF 程序的类型和探针的类型。否则，我们需要手动附加探针：

```
1 link = bpf_program__attach_kprobe(prog, false, "vfs_read");
2 link = bpf_program__attach_tracepoint(prog, "syscalls", "sys_enter_execve");
```

探针附加完成后，我们可以获取 BPF 程序中的映射，方便后面的数据读取：

```
1 int fd = bpf_object__find_map_fd_by_name(obj, "my_map");
```

这里的第二个参数即是 BPF 程序中定义的映射的名字。

当 BPF 程序附加后，如没有任何错误，那么就已经开始执行。我们可以让程序稍微运行一段时间，让映射积累足够的数据，等用户按下 `^C` 再获取映射中的数据、打印、退出。以 `kprobe/vfs_read` 为例，其用户空间程序 `user.c` 代码如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <errno.h>
5 #include <string.h>  // sleep()
```

```c
#include <unistd.h>
#include <bpf/libbpf.h>
#include <bpf/bpf.h>

struct map_key {
    char comm[16];
    unsigned int pid;
};

static int fd;

static void handler(int sig)
{
    struct map_key key, next_key;
    unsigned int count;
    printf("%-8s %-16s %-5s\n", "PID", "COMM", "COUNT");
    /*
        给定一个键 key，bpf_map_get_next_key() 会将下一个键存放到第三个参数 next_key，
        之后调用 bpf_map_lookup_elem() 通过键获取值。注意两个函数的参数都是使用的地址。
        当没有更多的键时，bpf_map_get_next_key() 返回非零，循环结束。
    */
    while (bpf_map_get_next_key(fd, &key, &next_key) == 0) {
            bpf_map_lookup_elem(fd, &next_key, &count);
            printf("%-8d, %-16s %-5d\n", key.pid, key.comm, count);
            key = next_key; // 将 key 设置成下一个，继续循环
    }
}

int main()
{
    struct bpf_object *obj;
    struct bpf_program *prog;
    struct bpf_link *link;

    obj = bpf_object__open(kern.o);
    prog = bpf_object__find_program_by_title(obj, "kprobe/vfs_read");
    bpf_object__load(obj)
    link = bpf_program__attach(prog);
    fd = bpf_object__find_map_fd_by_name(obj, "my_map");

    signal(SIGINT, handler);
    printf("Press ^C to stop\n");

    sleep(99999);

    bpf_link__destroy(link);
    bpf_object__close(obj);

    return 0;
}
```

这里我们定义了一个信号处理函数 handler(),并将其注册到 SIGINT,这样一来用户按下 ^C 时 handler() 就会执行，函数内部通过 bpf_map_get_next_key() 和 bpf_map_lookup_elem() 从映射中读取数据并打印。存储映射描述符的 fd 是一个全局变量，方便 handler() 访问。

struct map_key 需和 kern.c 中定义的保持一致，否则不能获取准确的数据。为使代码简洁我们可以将其放在一个新的头文件 common.h 中，并在 kern.c 和 user.c 中包含。

使用 clang 编译 user.c:

```
$ clang user.c -o user.o -L /lib/ -lbpf
```

运行:

```
$ sudo ./prog
```

```
PID     COMM            COUNT
1130    gdbus           3
1479    conky           7
2120    gdbus           7
1288    gdbus           3
1475    conky           1
1224    kdeconnectd     3
1283    ibus-engine-lib 4
18004   electron        2
[...]
```

§5 BCC

BCC 的部分 API 拥有和 libbpf 相同的逻辑。我们可以使用 libbpf 的 API 打开 BPF 对象文件、获取 BPF 程序和映射，再使用 BCC 的 API 操作映射。但这样一来并不如直接使用 libbpf。我们这里关心的是 BCC 比 libbpf 更为抽象、方便的地方:

- 我们可以把 BPF 源代码作为参数传给 BCC，BCC 会将其动态编译成 BPF 程序，完成这一步的主要是 bcc/bcc_common.h 中的 bpf_module_create_c() 和 bpf_module_create_c_from_string()。
- BCC 提供了语法糖来简化映射的声明。我们不必通过 struct bpf_map_def 定义映射，而是直接用 BPF_HASH(my_map) 定义名为 map_map 的哈希映射，用 BPF_ARRAY() 定义数列映射等。这样定义的映射包含对自身进行操作的方法，所以我们可以不再用 BPF 帮助函数。
- BCC 简化了对结构体的解引用。为了确保 BPF 程序访问内核地址的安全，BPF 要求使用诸如 bpf_probe_read() 一类的帮助函数来读取特定位置的内核数据，但当需要访问的数据需要多次结构体的解引用 (如 a->b->c->d) 才能得到时，我们也必须多次使用前述的帮助函数。BCC 内部会自动将普通的结构解引用自动重写为调用帮助函数的形式，从而简化开发。

和上一节一样，我们先看看 BCC 中常用的 API:

```
1 // 从文件动态编译
2 #include <bcc/bcc_common.h>
3 void * bpf_module_create_c(const char *filename, unsigned flags,
4                            const char *cflags[], int ncflags,
5                            bool allow_rlimit, const char *dev_name);
6
7 // 从字符串动态编译
```

```
void * bpf_module_create_c_from_string(const char *text, unsigned flags,
                                       const char *cflags[], int ncflags,
                                       bool allow_rlimit, const char *dev_name);

// 关闭上面函数返回的 BPF 模块
void bpf_module_destroy(void *program);

// 获取 BPF 程序的指针
void * bpf_function_start(void *program, const char *name);

// 获取 BPF 程序的大小
size_t bpf_function_size(void *program, const char *name);

// 加载程序
int bcc_func_load(void *program, int prog_type, const char *name,
                  const struct bpf_insn *insns, int prog_len,
                  const char *license, unsigned kern_version,
                  int log_level, char *log_buf, unsigned log_buf_size,
                  const char *dev_name);

// 根据映射名获取其描述符
int bpf_table_fd(void *program, const char *table_name);

// 附加探针
#include <bcc/libbpf.h>
int bpf_attach_kprobe(int progfd, enum bpf_probe_attach_type attach_type,
                      const char *ev_name, const char *fn_name,
                      uint64_t fn_offset, int maxactive);

int bpf_attach_tracepoint(int progfd, const char *tp_category,
                          const char *tp_name);

// 映射操作
int bpf_update_elem(int fd, void *key, void *value, unsigned long long flags);
int bpf_lookup_elem(int fd, void *key, void *value);
int bpf_delete_elem(int fd, void *key);
int bpf_get_first_key(int fd, void *key, size_t key_size);
int bpf_get_next_key(int fd, void *key, void *next_key);
int bpf_lookup_and_delete(int fd, void *key, void *value);
```

基于 BCC，我们可以把前面探测的 `vfs_read` 的程序改写成以下形式。其中用到了 `BPF_HASH()` 定义了一个名为 `my_map`，键和值分别为 `struct map_key` 和 `u32` 且能容纳 1024 个项目的哈希映射。

```
struct map_key {
    char comm[16];
    unsigned int pid;
};

BPF_HASH(my_map, struct map_key, u32, 1024);

int vfs_read_probe(struct pt_regs *ctx)
```

```
{
    struct map_key k;
    u32 init = 1;
    u32 *res;

    k.pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&k.comm, sizeof(k.comm));

    res = my_map.lookup(&k);
    if(res) {
        (*res)++;
    } else {
        my_map.update(&k, &init);
    }

    return 0;
}
```

将上面的代码保存为 kern.c，但不进行编译。下面是用户空间的代码。使用 BCC 模糊了 BPF 程序和用户程序的界线，只有单个程序，我们将其命令为 prog.c：

```
#include <stdio.h>
/*
enum bpf_probe_attach_type {
    BPF_PROBE_ENTRY,
    BPF_PROBE_RETURN
};
*/
#include <bcc/libbpf.h>
#include <bcc/bcc_common.h>
#include <linux/bpf.h> // enum bpf_prog_type
#include <signal.h>
#include <unistd.h>

struct map_key {
    char comm[16];
    unsigned int pid;
};

static volatile bool loop = true;

static void handler(int sig)
{
    loop = false;
}

static void print_result(int map)
{
    struct map_key key, next_key;
    unsigned int count;
    printf("\n%-8s %-16s %-5s\n", "PID", "COMM", "COUNT");
```

```
31
32      if(bpf_get_first_key(map, &key, sizeof(struct map_key)))
33          return;
34
35      while(bpf_get_next_key(map, &key, &next_key) == 0) {
36          bpf_lookup_elem(map, &key, &count);
37          printf("%-8d %-16s %-5d\n", key.pid, key.comm, count);
38          key = next_key;
39      }
40      exit(0);
41  }
42
43  int main()
44  {
45      char filename[] = "kern.c";
46
47      void *prog = bpf_module_create_c(filename, 0, NULL, 0, 1, NULL);
48      void *func = bpf_function_start(prog, "vfs_read_probe");
49
50      int fd = bcc_func_load(prog, BPF_PROG_TYPE_KPROBE, "vfs_read_probe",
51                             func, bpf_function_size(prog, "vfs_read_probe"),
52                             bpf_module_license(prog),
53                             bpf_module_kern_version(prog),
54                             0, NULL, 0, NULL);
55
56      bpf_attach_kprobe(fd, BPF_PROBE_ENTRY, "vfs_read", "vfs_read", 0, 0);
57
58      signal(SIGINT, handler);
59      printf("Probing... Press ^C to stop\n");
60
61      while (loop) {
62          sleep(1);
63      }
64
65      int map = bpf_table_fd(prog, "my_map");
66      print_result(map);
67
68      return 0;
69  }
```

对于前述 tracepoint 程序来说，我们需要 56 行的 `bpf_attach_kprobe()` 改为：

```
1  bpf_attach_tracepoint(fd, "syscalls", "sys_enter_execve");
```

之后 `print_result()` 也需要做相应的修改。

编译命令如下：

```
$ clang prog.c -o prog -L /lib/ -lbcc
```

§6 Python BCC

Python 的 BCC 绑定除了包装 BCC 的一堆 API 以外，还提供了一些方便的特性。在 BPF 程序的编写方面，使用 Python BCC 和 BCC 并无二异，只不过在 Python 中我们使用面向对象的范式，主要的类是 `BPF` 类。这一节我们主要关注 BCC 函数签名的语法糖和映射数据的获取。

```python
#! /bin/python
from bcc import BPF
from time import sleep

bpf = BPF(text="""
struct map_key {
    char comm[16];
    unsigned int pid;
};

BPF_HASH(my_map, struct map_key, u32, 1024);

int kprobe__vfs_read(struct pt_regs *ctx)
{
    struct map_key k;
    u32 init = 1;
    u32 *res;

    k.pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&k.comm, sizeof(k.comm));

    res = my_map.lookup(&k);
    if(res) {
        (*res)++;
    } else {
        my_map.update(&k, &init);
    }

    return 0;
}
""")

# bpf.attach_kprobe(event="vfs_read", fn_name="vfs_read_probe")
def print_result():
    print("%-32s %-8s %-8s" % ("COMM", "PID", "COUNT"))
    for k, v in bpf["my_map"].items():
        print("%-32s %-8d %-8s" % (k.comm.decode("utf-8"), k.pid, v.value))

print("Tracing, press ^C to stop...")
while True:
    try:
        sleep(99999)
    except KeyboardInterrupt:
        print()
        print_result()
```

```
46          break
```

第 5 行实例化了一个 BPF 类，其 text 属性包含了 BPF 程序的源码。源码和上一节的一样，这里不再阐述。实例化之后，BPF 类的初始化方法 __init__() 会自动将 BPF 源码编译。

在 Python BCC 中，遵循一定签名规范的 BPF 函数可以被自动附加到相应的被测函数，所以这里不需要 33 行的代码。这里的 kprobe__vfs_read() 就表示使用 kprobe 追踪 vfs_read()，其在编译后会自动附加到 vfs_read()。如果是 tracepoint，则先前的 BPF 函数可以改写成：

```
1 TRACEPOINT_PROBE(syscalls, sys_enter_execve) {
2     [...]
3 }
```

这种情况下被测函数的参数通过固定的变量 args 获取，如 args->filename。

由于 Python 支持迭代器，所以从 BPF 类的实例中获取映射的数据不需要在 while 中反复循环，而是直接调用 items() 方法。于是在 print_result() 函数中，我们通过 bpf["my_map"] 得到 BPF 程序中的映射 (在 Python 中被包装成了一个类)，再调用其 items() 方法获取所有的键和值并迭代输出。

§7 Ring Buffer

在前面的例子中我们都是主动地在用户空间从映射中读取数据。这一节，我们讨论 BPF 提供的另外一种让用户空间程序被动地处理数据的数据结构——ring buffer，简称 ringbuf。熟悉 BPF 的读者可能了解，现在 BCC 和 libbpf 项目中的示例大多基于 perfbuf。perfbuf 是另一种使用户空间和 BPF 程序快速通信的数据结构，两者利用了在用户空间定义的遵循特定签名的回调函数来处理数据：一旦 BPF 程序调用对应的 BPF 帮助函数提交了搜集的数据，回调函数就会被调用，我们可以在回调函数内处理 BPF 程序提交的数据。

ringbuf 相较于 perfbuf 属于后来者，且后来居上。ringbuf 在性能方面的表现比 perfbuf 更优秀，并且 ringbuf 还解决了一些 perfbuf 存在的问题，如内存的开销，数据的乱序等等。Andrii Nakryiko 的 BPF ring buffer 一文 [1] 在这方面已经作了解释，这里就不再展开。

在 BPF 程序中 ringbuf 的定义可以如下：

```
1 struct bpf_map_def SEC("maps") my_map = {
2     .type = BPF_MAP_TYPE_RINGBUF,
3     .max_entries = 256 * 1024
4 };
```

这里我们只需要指定 ringbuf 的容量 (max_entries)，注意这里的单位是字节，且值需要是内核页 (一般为 4KiB) 的倍数。

和 ringbuf 相关的 BPF 帮助函数有以下几个：

```
1 long bpf_ringbuf_output(void *ringbuf, void *data, u64 size, u64 flags);
2 void *bpf_ringbuf_reserve(void *ringbuf, u64 size, u64 flags);
3 void bpf_ringbuf_submit(void *data, u64 flags);
4 void bpf_ringbuf_discard(void *data, u64 flags);
5 u64 bpf_ringbuf_query(void *ringbuf, u64 flags);
```

这里我们主要关心前三个函数：

- `bpf_ringbuf_output()` 和 `bpf_perf_event_output()` 类似，用于将数据放进缓冲区，供用户空间处理。
- `bpf_ringbuf_reserve()` 和下一个函数是 ringbuf 相较于 perfbuf 的优势所在。当被测函数被高频调用、或者用户空间的回调函数处理数据太慢，则 ringbuf 会装满，导致 `bpf_ringbuf_output()` 执行失败，这样一来就浪费了额外的 CPU 时钟。所以如果能提前知道 ringbuf 是否已经装满，我们就可以提前判断是否还要继续向里面添加数据。如果 ringbuf 已装满，那么 `bpf_ringbuf_reserve()` 会返回 `NULL`，于是本次 BPF 程序就可以返回而不执行后续代码。
- `bpf_ringbuf_submit()` 在 `bpf_ringbuf_reserve()` 返回一个有效的地址后调用，用来向 ringbuf 传递数据。

将前面的 tracepoint 程序改写为使用 ringbuf 的形式：

```c
#define __TARGET_ARCH_x86

#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include "common.h"

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_RINGBUF,
    .max_entries = 256 * 1024
};

struct execve_args {
    u64 unused;
    int sys_nr;
    u64 *filename;
    u64 *argv;
    u64 *envp;
};

SEC("tracepoint/syscalls/sys_enter_execve")
int prog(struct execve_args *ctx)
{
    struct data d;
    d.pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&d.comm, sizeof(d.comm));
    bpf_probe_read_user_str(&d.filename, sizeof(d.filename), ctx->filename);

    bpf_ringbuf_output(&my_map, &d, sizeof(d), 0);

    return 0;
}

char _license[] SEC("license") = "GPL";
```

在第 30 行我们使用了性能较低的 bpf_ringbuf_output()。有了前面关于 server() 和 submit() 的讨论，我们可以进一步将 prog() 函数改为：

```c
SEC("tracepoint/syscalls/sys_enter_execve")
int prog(struct execve_args *ctx)
{
    struct data *d;
    d = bpf_ringbuf_reserve(&my_map, sizeof(*d), 0);
    if (!d) {
        return 0;
    }

    d->pid = (u32)bpf_get_current_pid_tgid();
    bpf_get_current_comm(&d->comm, sizeof(d->comm));
    bpf_probe_read_user_str(&d->filename, sizeof(d->filename), ctx->filename);

    bpf_ringbuf_submit(d, 0);

    return 0;
}
```

在第 5 到 8 行中我们判断 ringbuf 中是否还有足够的空间，如果没有，直接退出，否则填充 struct data 并通过 bpf_ringbuf_submit() 发送到 ringbuf。

下面我们看看 libbpf 中常用的关于 ringbuf 的 API：

```c
// 回调函数的签名
typedef int (*ring_buffer_sample_fn)(void *ctx, void *data, size_t size);

// 从映射描述符创建一个 ringbuf
LIBBPF_API struct ring_buffer *
ring_buffer__new(int map_fd, ring_buffer_sample_fn sample_cb, void *ctx,
                 const struct ring_buffer_opts *opts);

// 轮询 ringbuf
LIBBPF_API int ring_buffer__poll(struct ring_buffer *rb, int timeout_ms);

// 释放 ringbuf 资源
LIBBPF_API void ring_buffer__free(struct ring_buffer *rb);
```

在用户空间层面，为了从 ringbuf 中读取数据，我们先用 ring_buffer__new() 创建一个 ring_buffer 结构。这个函数的参数包含了一个函数指针 sample_cb，其签名在上方已经给出，三个参数分别代表事件的上下文、ringbuf 中的数据以及数据的大小。ring_buffer__new() 的后两个参数一般设为 NULL。

ring_buffer__poll() 用来开启轮询。这样做的效果是，每当 BPF 程序调用 bpf_ringbuf_output() 或 bpf_ringbuf_submit() 提交了一份数据，我们创建 ringbuf 时传入的 sample_cb 就会被调用，其三个参数也就会被赋予相应的值。

遵循第4节中 user.c 的逻辑，我们只需要自定义一个回调函数 handler_event()，并使用相关的 ringbuf API 即可。部分代码如下：

```c
static volatile bool loop = true;

static void sig_handler(int sig)
{
    loop = false;
}

static int event_handler(void *ctx, void *data, size_t size)
{
    /* 需要将 data 指针转换成相应的类型 */
    struct data *e = data;
    printf("%-16s %-7u %-32s\n", e->comm, e->pid, e->filename);

    return 0;
}

int main()
{
    struct ring_buffer *ringbuf;
    int err;
    [...]

    signal(SIGINT, sig_handler);
    printf("Press ^C to stop\n");
    printf("%-16s %-7s %-32s\n", "COMM", "PID", "FILENAME");

    ringbuf = ring_buffer__new(fd, event_handler, NULL, NULL);
    if (!ringbuf) {
        fprintf(stderr, "ERROR: creating ring buffer failed\n");
        goto cleanup;
    }

    /* 当用户按下 ^C 后 loop 为 false，跳出循环 */
    while (loop) {
        /* 开启轮询 */
        err = ring_buffer__poll(ringbuf, 100);
        if (err == -EINTR) {
            err = 0;
            break;
        }
        if (err < 0) {
            printf("Error polling ring buffer: %d\n", err);
            break;
        }
    }

    [...]
    ring_buffer__free(ringbuf);

    return 0;
    [...]
```

```
52 }
```

BCC 中和 ringbuf 相关的 API 和 libbpf 的相仿，如下：

```
typedef int (*ring_buffer_sample_fn)(void *ctx, void *data, size_t size);
void * bpf_new_ringbuf(int map_fd, ring_buffer_sample_fn sample_cb, void *ctx);
void bpf_free_ringbuf(struct ring_buffer *rb);
int bpf_poll_ringbuf(struct ring_buffer *rb, int timeout_ms);
```

上述 API 的用法和 Python BCC 的 ringbuf API 这里就不再给出，对后者感兴趣的读者可参考其文档。

§8 原理

编写 BPF 程序时指定的 `SEC()` 是怎样发挥作用的？BCC 中创建映射的语法糖又是什么原理？为什么在 Python 中只需要给 `BPF` 类传递一份源码，之后便可以直接读取数据？本节会有选择性地分析这些工具的原理。

8.1 相关的系统调用

BCC 和 libbpf 中几乎所有对 BPF 程序和映射的操作最后都归结为对 `bpf()`、`perf_event_open()` 和 `ioctl()` 这三个系统调用的使用。

```
#include <linux/bpf.h>

int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

`bpf()` 系统调用主要用来直接操作映射或者加载 BPF 程序。`cmd` 是发送给 `bpf()` 的命令，`attr` 是一个联合体，可以存储和映射或者加载 BPF 程序相关的数据，`size` 表示 `attr` 的大小。具体的细节 *bpf(2)* 已经给出，这里不再赘述。

```
#include <linux/perf_event.h>
#include <linux/hw_breakpoint.h>

int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu, int group_fd,
                    unsigned long flags);
```

`perf_event_open()` 用于监控各种软硬件事件，以及创建 tracepoint、kprobe 和 uprobe。通过 `bpf()` 加载后的 BPF 程序并没有工作，需要将其和 `perf_event_open()` 创建的相应的事件联系起来——也就是前文所讲的探针 (BPF 程序) 的附加，BPF 程序才开始在被测函数被调用前或后被调用。`perf_event_open()` 的详细用法参照 *perf_event_open(2)*。

```
#include <stropts.h>

int ioctl(int fildes, int request, ...);
```

最后，通过将前面运行完 bpf() 和 perf_event_open() 后得到的两个返回值以及适当的指令作为参数传递给 ioctl() 完成探针的附加。

8.2 libbpf

libbpf 的源码在 Linux 源码的 tools/lib/bpf/目录下，其中，我们感兴趣的主要是 bpf.[ch][4]和 libbpf.[ch] 四个文件。

bpf.[ch] 主要是对上述 bpf() 系统调用的封装。下面的两个函数分别包装了 bpf() 系统调用本身，以及其加载程序的功能。

```
static inline int sys_bpf(enum bpf_cmd cmd, union bpf_attr *attr,
unsigned int size)
{
    return syscall(__NR_bpf, cmd, attr, size);
}

static inline int sys_bpf_prog_load(union bpf_attr *attr, unsigned int size)
{
    int retries = 5;
    int fd;

    do {
        fd = sys_bpf(BPF_PROG_LOAD, attr, size);
    } while (fd < 0 && errno == EAGAIN && retries-- > 0);

    return fd;
}
```

又如，先前使用的更新 BPF 映射的函数 bpf_map_update_elem() 定义如下：

```
int bpf_map_update_elem(int fd, const void *key, const void *value,
                        __u64 flags)
{
    union bpf_attr attr;

    memset(&attr, 0, sizeof(attr));
    attr.map_fd = fd;
    attr.key = ptr_to_u64(key);
    attr.value = ptr_to_u64(value);
    attr.flags = flags;

    return sys_bpf(BPF_MAP_UPDATE_ELEM, &attr, sizeof(attr));
}
```

其中 libbpf 自动帮忙构建了 bpf() 需要的 attr 参数，作为用户只需要关心映射本身的键值。其余的映射操作函数也均在构建好适当的参数后调用 sys_bpf()。

打开 BPF 对象文件用到的 bpf_object__open() 函数实现在 libbpf.c，其调用链如下：

[4]表示 bpf.c 和 bpf.h。

```
bpf_object__open()
    bpf_object__open_xattr()
        __bpf_object__open_xattr()
            __bpf_object__open()
                    bpf_object__new()
                    bpf_object__elf_init()
                    bpf_object__elf_collect()
                    bpf_object__init_maps()
                    bpf_object__for_each_program()
```

每一个 BPF 对象文件对应一个 libbpf 中定义的 `struct bpf_object`,所以可以看到有 `bpf_object__new()` 来创建一个 `struct bpf_object`。

`bpf_object__elf_init()` 分析 BPF 对象文件的结构,如 ELF 头、节区名等。`bpf_object__elf_collect()` 扫描搜集有特定区块名的 ELF 区块，如 `license`、`version`、`maps` 和`.text`，部分代码如下：

```
while ((scn = elf_nextscn(elf, scn)) != NULL) {
    idx++;
    [...]
    if (strcmp(name, "license") == 0) {
        err = bpf_object__init_license(obj, data->d_buf, data->d_size);
        if (err)
        return err;
    } else if (strcmp(name, "version") == 0) {
        err = bpf_object__init_kversion(obj, data->d_buf, data->d_size);
        if (err)
        return err;
    } else if (strcmp(name, "maps") == 0) {
        obj->efile.maps_shndx = idx;
    } else if (strcmp(name, MAPS_ELF_SEC) == 0) {
        obj->efile.btf_maps_shndx = idx;
    } else if (strcmp(name, BTF_ELF_SEC) == 0) {
       btf_data = data;
    } else if (strcmp(name, BTF_EXT_ELF_SEC) == 0) {
       btf_ext_data = data;
    } else if (sh.sh_type == SHT_SYMTAB) {
        /* already processed during the first pass above */
    } else if (sh.sh_type == SHT_PROGBITS && data->d_size > 0) {
       if (sh.sh_flags & SHF_EXECINSTR) {
           if (strcmp(name, ".text") == 0)
            obj->efile.text_shndx = idx;
            err = bpf_object__add_programs(obj, data, name, idx);
            if (err)
           return err;
[...]
```

`bpf_object__init_maps()` 初始化我们在 BPF 对象文件中用 `struct bpf_map_def` 定义的映射，即将 `struct bpf_map_def` 中的信息拷贝到 `struct bpf_object` 的 `maps` 字段中，供加载 BPF 程序时创建映射。

`bpf_object__for_each_program()` 是一个宏，用来遍历一个 BPF 对象文件中的 BPF 程序，在 libbpf 中表现为遍历 `struct bpf_object` 中的 `programs` 字段。在前面的代码链中的 `__bpf_object__open()`

里，bpf_object__for_each_program() 的使用如下：

```
bpf_object__for_each_program(prog, obj) {
    prog->sec_def = find_sec_def(prog->sec_name);
    if (!prog->sec_def) {
        /* couldn't guess, but user might manually specify */
        pr_debug("prog '%s': unrecognized ELF section name '%s'\n",
        prog->name, prog->sec_name);
        continue;
    }

    if (prog->sec_def->is_sleepable)
    prog->prog_flags |= BPF_F_SLEEPABLE;
    bpf_program__set_type(prog, prog->sec_def->prog_type);
    bpf_program__set_expected_attach_type(prog,
    prog->sec_def->expected_attach_type);

    if (prog->sec_def->prog_type == BPF_PROG_TYPE_TRACING ||
    prog->sec_def->prog_type == BPF_PROG_TYPE_EXT)
    prog->attach_prog_fd = OPTS_GET(opts, attach_prog_fd, 0);
}
```

这里 find_sec_def() 通过利用 section_defs 这个常量数组比对区块名并决定 BPF 程序的类型，之后 bpf_program__set_type() 和 bpf_program__set_expected_attach_type() 又通过前面获取的数据设置程序的类型和附加类型。

将 BPF 程序加载进内核的函数 bpf_object__load() 调用链如下：

```
bpf_object__load()
    bpf_object__load_xattr()
        bpf_object__create_maps()
        bpf_object_load_progs()
            bpf_program__load()
                load_program()
                    libbpf__bpf_prog_load() (bpf.c)
                        sys_bpf_prog_load()
```

bpf_object__create_maps() 负责根据给定 struct bpf_object 中的 maps 字段创建映射。之后 bpf_object_load_progs() 会循环将 BPF 对象文件中的所有 BPF 程序加载进内核。最后调用链触及 sys_bpf_prog_load()，并最终调用 bpf()。

前面说过，bpf_program__attach() 可以依靠节区名智能附加探针，其定义如下：

```
struct bpf_link *bpf_program__attach(struct bpf_program *prog)
{
    const struct bpf_sec_def *sec_def;

    sec_def = find_sec_def(prog->sec_name);
    if (!sec_def || !sec_def->attach_fn)
    return ERR_PTR(-ESRCH);

    return sec_def->attach_fn(sec_def, prog);
}
```

其中也用到了上面提到的 find_sec_def()。根据节区名的不同，sec_def->attach_fn 也会是不同的函数指针，对应附加不同类型的 BPF 程序的函数。和 kprobe 相关的元素在 section_defs 数组中的表示为：

```
1 SEC_DEF("kprobe/", KPROBE, .attach_fn = attach_kprobe)
```

对应的 attach_fn 是 attach_kprobe,而 attach_kprobe() 又进一步调用了 bpf_program__attach_kprobe(),即 libbpf 中附加 kprobe 探针的函数。

依然以 kprobe 为例，我们可以使用 bpf_program__attach_kprobe()，其在 libbpf.c 的调用链如下：

```
bpf_program__attach_kprobe()
    perf_event_open_probe()
        perf_event_open()
    bpf_program__attach_perf_event()
        ioctl()
```

程序首先通过 perf_event_open() 得到了一个和给定 kprobe 相关的描述符，之后再通过 ioctl() 将对应的 BPF 程序和 kprobe 联系起来，之后 BPF 程序便开始工作。

bpf_program__attach_tracepoint() 的调用链和 bpf_program__attach_kprobe()，这里不再讨论。

8.3 BCC

BCC 的 C 库的代码在 BCC 项目的 src/cc 目录下，和映射操作、程序附加、ringbuf 相关的 API 均实现在 libbpf.c 中，和 perfbuf 相关的代码在 perf_reader.[ch] 中。

动态编译 BPF 程序的函数 bpf_module_create_c_from_string() 实现在 bcc_common.cc。函数中实例化了一个 BPFModule 对象 (bcc_module.[ch])。之后这个对象的 load_string() 方法被调用，其内部又进一步调用了同一源文件中的 load_cfile()。总的函数调用关系如下：

```
bpf_module_create_c_from_string()
    BPFModule::load_string()
        BPFModule::load_cfile()
            ClangLoader::parse()
                ClangLoader::do_compile()
```

ClangLoader 类的定义和实现分别在 frontends/clang/loader.h 和 frontends/clang/loader.cc。parse() 中构建了传递给 clang 的命令行参数。do_compile() 利用 LLVM/clang 进行 BPF 程序的编译。

export/helpers.h 包含了 BCC 语法糖的定义，涉及映射创建、struct pt_regs 的访问以及 BPF 帮助函数等。这个头文件仅供 BCC 内部使用。

说通过 BCC 语法糖创建映射实际上不严谨。这样的语法糖展开后实际上是一个结构体的定义，其中的字段描述了映射的参数。BCC 会通过这个结构体的字段构造一个映射，相关的代码在 frontends/clang/b_frontend_action.cc。另外，在 BPF 源码中使用语法糖创建的“映射”的方法调用、结构体的连续解引用等都会被这个文件的代码重写为调用 BPF 帮助函数的形式。

附加 kprobe 探针的 bpf_attach_kprobe() 调用链如下：

```
bpf_attach_kprobe()
    bpf_attach_probe()
        bpf_try_perf_event_open_with_probe() 或 create_probe_event()
            perf_event_open()
```

```
        bpf_attach_tracing_event()
                ioctl()
```

和 libbpf 中对应的函数逻辑相似，`bpf_attach_kprobe()` 最后也用到了 `perf_event_open()` 和 `ioctl()`。只不过，如果 `bpf_try_perf_event_open_with_probe()` 调用失败，BCC 还会直接使用 debugfs 进行探针的创建。

BCC 操作映射的 API 均调用了 libbpf 中的 API，这里不再赘述。

8.4 Python BCC

Python BCC 的源码在 BCC 项目的 `src/python/bcc` 目录下。Python BCC 通过 Python 的 `ctypes` 将 BCC 库 `libbcc` 封装起来，相关代码在 `libbcc.py`。

```
import ctypes as ct

lib = ct.CDLL("libbcc.so.0", use_errno=True)
```

Python BCC 中主要的元素是 `BPF` 类，其定义在 `__init__.py`。在 `BPF` 类的初始化方法 `__init__()` 中可以找到利用 BCC 的 `bpf_module_create_c_from_string()` 动态编译 BPF 源码的代码：

```
self.module = lib.bpf_module_create_c_from_string(text,
                                                   self.debug,
                                                   cflags_array, len(cflags_array),
                                                   allow_rlimit, device)
```

其中 `text` 为 `BPF` 的属性，包含 BPF 程序源码，一般在实例化 `BPF` 类时给出。

`BPF` 类中探针附加的函数也都利用了 BCC 相关的函数，如下：

```
def attach_kprobe(self, event=b"", event_off=0, fn_name=b"", event_re=b""):
    [...]
    fn = self.load_func(fn_name, BPF.KPROBE)
    ev_name = b"p_" + event.replace(b"+", b"_").replace(b".", b"_")
    fd = lib.bpf_attach_kprobe(fn.fd, 0, ev_name, event, event_off, 0)
    if fd < 0:
        raise Exception("Failed to attach BPF program %s to kprobe %s" %
                        (fn_name, event))
    self._add_kprobe_fd(ev_name, fd)
    return self

def attach_tracepoint(self, tp=b"", tp_re=b"", fn_name=b""):
    [...]
    fn = self.load_func(fn_name, BPF.TRACEPOINT)
    (tp_category, tp_name) = tp.split(b':')
    fd = lib.bpf_attach_tracepoint(fn.fd, tp_category, tp_name)
    if fd < 0:
        raise Exception("Failed to attach BPF program %s to tracepoint %s" %
                        (fn_name, tp))
    self.tracepoint_fds[tp] = fd
    return self
```

Python BCC 之所以支持自动附加拥有适当签名的 BPF 函数，是因为在 `BPF` 类的初始化方法的最后有一句：

```
self._trace_autoload()
```

而 `_trace_autoload()` 的代码如下：

```
def _trace_autoload(self):
    for i in range(0, lib.bpf_num_functions(self.module)):
        func_name = lib.bpf_function_name(self.module, i)
        if func_name.startswith(b"kprobe__"):
            fn = self.load_func(func_name, BPF.KPROBE)
            self.attach_kprobe(
                event=self.fix_syscall_fnname(func_name[8:]),
                fn_name=fn.name)
        elif func_name.startswith(b"kretprobe__"):
            fn = self.load_func(func_name, BPF.KPROBE)
            self.attach_kretprobe(
                event=self.fix_syscall_fnname(func_name[11:]),
                fn_name=fn.name)
        elif func_name.startswith(b"tracepoint__"):
            fn = self.load_func(func_name, BPF.TRACEPOINT)
            tp = fn.name[len(b"tracepoint__"):].replace(b"__", b":")
            self.attach_tracepoint(tp=tp, fn_name=fn.name)
        elif func_name.startswith(b"raw_tracepoint__"):
            fn = self.load_func(func_name, BPF.RAW_TRACEPOINT)
            tp = fn.name[len(b"raw_tracepoint__"):]
            self.attach_raw_tracepoint(tp=tp, fn_name=fn.name)
        elif func_name.startswith(b"kfunc__"):
            self.attach_kfunc(fn_name=func_name)
        elif func_name.startswith(b"kretfunc__"):
            self.attach_kretfunc(fn_name=func_name)
        elif func_name.startswith(b"lsm__"):
            self.attach_lsm(fn_name=func_name)
```

其中，BCC 中的 `bpf_num_functions()` 搭配 `bpf_function_name()` 可以获得 BPF 函数的名字，之后的代码便开始通过 BPF 函数的名字来加载 BPF 程序以及附加探针。

使用 Python API 的方便之处在于对映射数据的获取。知道了映射的名称（如 `my_map`），则可以通过 `bpf["my_map"]` 获取这个映射的对象。这样访问映射的原理在于 `BPF` 类实现了一个 `__getitem__()` 魔术方法。在 Python 中，定义了这个方法的类支持以这种类似索引的方式返回数据。`__getitem__()` 在 `BPF` 类中的实现如下：

```
def __getitem__(self, key):
    if key not in self.tables:
        self.tables[key] = self.get_table(key)
    return self.tables[key]
```

可以看出，BPF 类的 tables 属性是一个包含了 BPF 程序中定义的映射的列表。如果正在请求的映射不在列表中，则调用 get_table() 新建一个映射对象。get_table() 的定义在 table.py，其内部根据请求的映射的类型返回对应的映射对象。这里说的映射对象实际上继承自 TableBase 类的子类，如 ArrayBase、RingBuf 等，定义均在 table.py。

映射对象都有一个 items() 方法,用来获取映射中的键和值,这也是为什么在前文中可以通过 bpf["my_map"].items() 获取数据。在 TableBase 中，items() 的调用关系如下：

```
items()
    iteritems()
        for key in self         # 对本类进行迭代，依赖 __iter__() 魔术方法
            Iter()              # 创建一个 Iter 类
                __next__()      # Iter 的方法
                next()          # Iter 的方法
                    next()      # TableBase 的方法
                        bpf_get_first_key() # BCC 中的方法
                        bpf_get_next_key()
```

§9 总结

无论是 libbpf 还是 BCC，BPF 程序的运行都会经历打开、加载、附加和关闭四个阶段，每个阶段都对应一个函数。BPF 对象文件打开后，位于不同节区的 BPF 程序、映射等信息被获取；加载操作将 BPF 程序注入内核并被验证，先前定义的映射也一并被创建，其描述符可以通过相关的函数获取从而可以在用户空间对其操作；加载后的 BPF 程序需要附加到指定的函数上才会生效；当用户空间的程序退出，相关的 BPF 程序、映射等资源均会自动释放。

由于 BCC 的体积较大，且基于 BCC 的工具需要在每次运行时利用 LLVM/clang 动态编译 BPF 程序，这会使得被测的系统在这一时间点发生改变；另外，由于有的 BPF 程序需要访问内核数据，而不同版本的内核的数据排布可能不一样，所以 BCC 要求被测系统的内核头文件存在，而这个要求不是每个系统都符合的。所以现在比较流行的、也是本文没有讲到的做法是使用 libbpf + BPF CO-RE。libbpf 依然是前面所讲的库，BPF CO-RE 指 “编译一次，到处运行 (Compile Conce - Run Everywhere)” 的 BPF 程序。其大致原理是将 BPF 程序编译成对象文件后，通过 bpftool 从 BPF 对象文件生成一个 “程序骨架”——实际上是一个 C 头文件，其中包含了和已经编写的 BPF 程序紧密相关的完成打开、加载、附加和关闭操作的函数，其背后利用 libbpf 的 API，这些函数的参数经过简化，可以方便地调用。感兴趣的读者可以参照 [1][12][11]。BCC 项目的 libbpf-tools 中包含基于 BPF CO-RE 的 BPF 工具示例。

Greg Marsden 的博客 [6] 详细讲述了不同类型 BPF 程序的使用场景、BPF 帮助函数、BPF 程序和用户空间的通信等。[5] 包含了和 BPF 底层技术细节相关的内容。关于 tracefs/ftrace 的介绍可以参考 [8]，[7] 介绍了怎样利用 tracefs 进行事件追踪。另外，tracefs 也支持基于 kprobe 的追踪 [9]。

bpftrace[3] 和 ply[10] 两个命令行程序定义了一套简单的语法来快速创建 BPF 程序用于系统追踪。性能分析大佬 Brendan Gregg 的书 [13] 中分门别类地介绍了基于 BCC 和 bptrace 的各种分析工具，他的博客 [4] 也包含了大量和性能分析、调优相关的内容。

最后，Awesome eBPF 项目 [2] 整合了大量关于 BPF 的资源，包括上面提到的部分。读者可以通过本项目快速建立对 BPF 的认识。

参考文献

[1] Andrii nakryiko 的博客. https://nakryiko.com/.

[2] Awesome ebpf. https://github.com/zoidbergwill/awesome-ebpf.

[3] bpftrace. https://github.com/iovisor/bpftrace.

[4] Brendan gregg 的博客. http://www.brendangregg.com/.

[5] cilium 文档. https://docs.cilium.io/en/stable/bpf/.

[6] Greg marsden 的博客. https://blogs.oracle.com/author/greg-marsdenurl.

[7] Linux 内核文档：event tracing. https://www.kernel.org/doc/html/latest/trace/events.html.

[8] Linux 内核文档：ftrace - function tracer. https://www.kernel.org/doc/html/latest/trace/ftrace.html.

[9] Linux 内核文档：kprobe-based event tracing. https://www.kernel.org/doc/html/latest/trace/kprobetrace.html.

[10] ply. https://github.com/iovisor/ply.

[11] Tips and tricks for writing linux bpf applications with libbpf. https://pingcap.medium.com/tips-and-tricks-for-writing-linux-bpf-applications-with-libbpf-404ca94daaee.

[12] Why we switched from bcc to libbpf for linux bpf performance analysis. https://pingcap.com/blog/why-we-switched-from-bcc-to-libbpf-for-linux-bpf-performance-analysis.

[13] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability.* Addison-Wesley Professional, 1st edition, 2019.