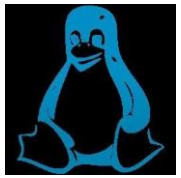
高效入门eBPF

西安邮电大学 贺东升

主办单位：Linux内核之旅开源社区

西安郵電大學
XI'AN UNIVERSITY OF POSTS & TELECOMMUNICATIONS



Linux内核之旅开源社区
Linux Kernel Travelopen Source Community


eBPF

BPF起源

- eBPF 是 extended BPF 的简称，而 BPF 的全称是 Berkeley Packet Filter, 即伯克利报文过滤器，它的设计思想来源于 1992 年的一篇论文“The BSD packet filter: A New architecture for user-level packet capture”（《BSD数据包过滤器：一种用于用户级数据包捕获的新体系结构》）。最初，BPF 是在 BSD 内核实现的，后来，由于其出色的设计思想，其他操作系统也将其引入，包括 Linux。

The BSD Packet Filter:
A New Architecture for User-level Packet Capture*

Steven McCanne† and Van Jacobson†
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720
mccanne@ee.lbl.gov, van@ee.lbl.gov

December 19, 1992

Abstract

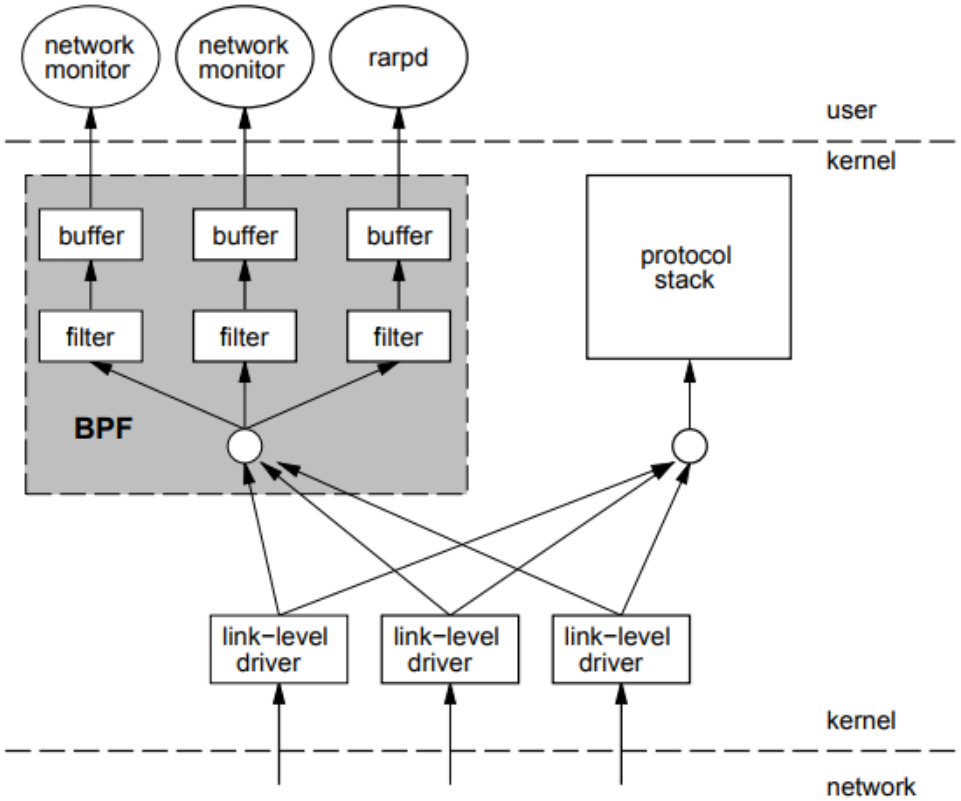
Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be

SunOS, the Ultrix Packet Filter[2] in DEC's Ultrix and Snoop in SGI's IRIX.

These kernel facilities derive from pioneering work done at CMU and Stanford to adapt the Xerox Alto 'packet filter' to a Unix kernel[8]. When completed in 1980, the CMU/Stanford Packet Filter, CSPF, provided a much needed and widely used facility. However on today's machines its performance, and

BPF功能

BPF架构原理图(来自论文原图):



BPF是作为内核报文传输路径的一个旁路存在的，当报文到达内核驱动程序后，内核在将报文上送协议栈的同时，会额外将报文的一个副本交给 BPF。之后，报文会经过BPF内部逻辑的过滤(这个逻辑可以自己设置)，然后最终送给用户程序(比如tcpdump)。

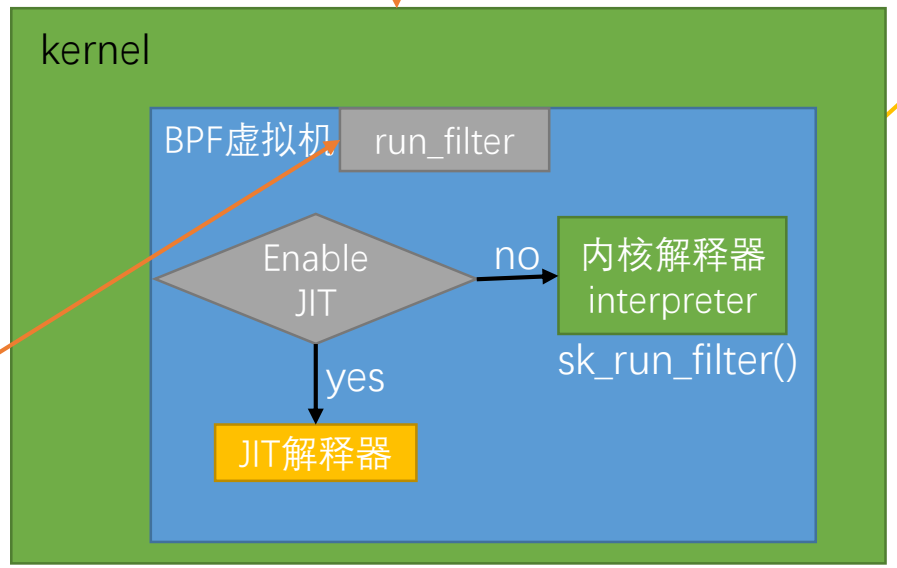
BPF的实现有哪些？

用户态自定义的过滤程序

```c
#include <netpacket/packet.h>
#include <linux/filter.h>
...

#define OP_LDH (BPF_LD  | BPF_H   | BPF_ABS)
#define OP_LDB (BPF_LD  | BPF_B   | BPF_ABS)
#define OP_JEQ (BPF_JMP | BPF_JEQ | BPF_K)
#define OP_RET (BPF_RET | BPF_K)

static struct sock_filter bpfcode[6] = {
    { OP_LDH, 0, 0, 12          },  // ldh [12]
    { OP_JEQ, 0, 2, ETH_P_IP    },  // jeq #0x800, L2, L5
    { OP_LDB, 0, 0, 23          },  // ldb [23]
    { OP_JEQ, 0, 1, IPPROTO_TCP },  // jeq #0x6, L4, L5
    { OP_RET, 0, 0, 0           },  // ret #0x0
    { OP_RET, 0, 0, -1          },  // ret #0xffffffff
};

int main(int argc, char **argv)
{
    ...
    sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

    if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, \
        &bpf, sizeof(bpf))) {
        perror("setsockopt ATTACH_FILTER");
        return 1;
    }
    ...
    return 0;
}
```

通过setsockopt()系统调用将BPF指令注入内核

```c
switch(optname) {
    case SO_ATTACH_FILTER:
        ...
        if (optlen == sizeof(struct sock_fprog)) {
            struct sock_fprog fprog;
            ...
            /* 从用户空间向内核空间拷贝BPF程序 */
            if (copy_from_user(&fprog, optval, sizeof(fprog)))
                break;
            /* 给 sock 挂载上BPF程序 */
            ret = sk_attach_filter(&fprog, sk);
        }
        break;

    case SO_DETACH_FILTER:
        /* 解挂BPF程序 */
        ret = sk_detach_filter(sk);
        break;

    default:
        ret = -ENOPROTOOPT;
        break;
    }
```



核心的实现就是模拟了一套简单的处理器，由累加器、索引寄存器、小块内存、PC 组成。

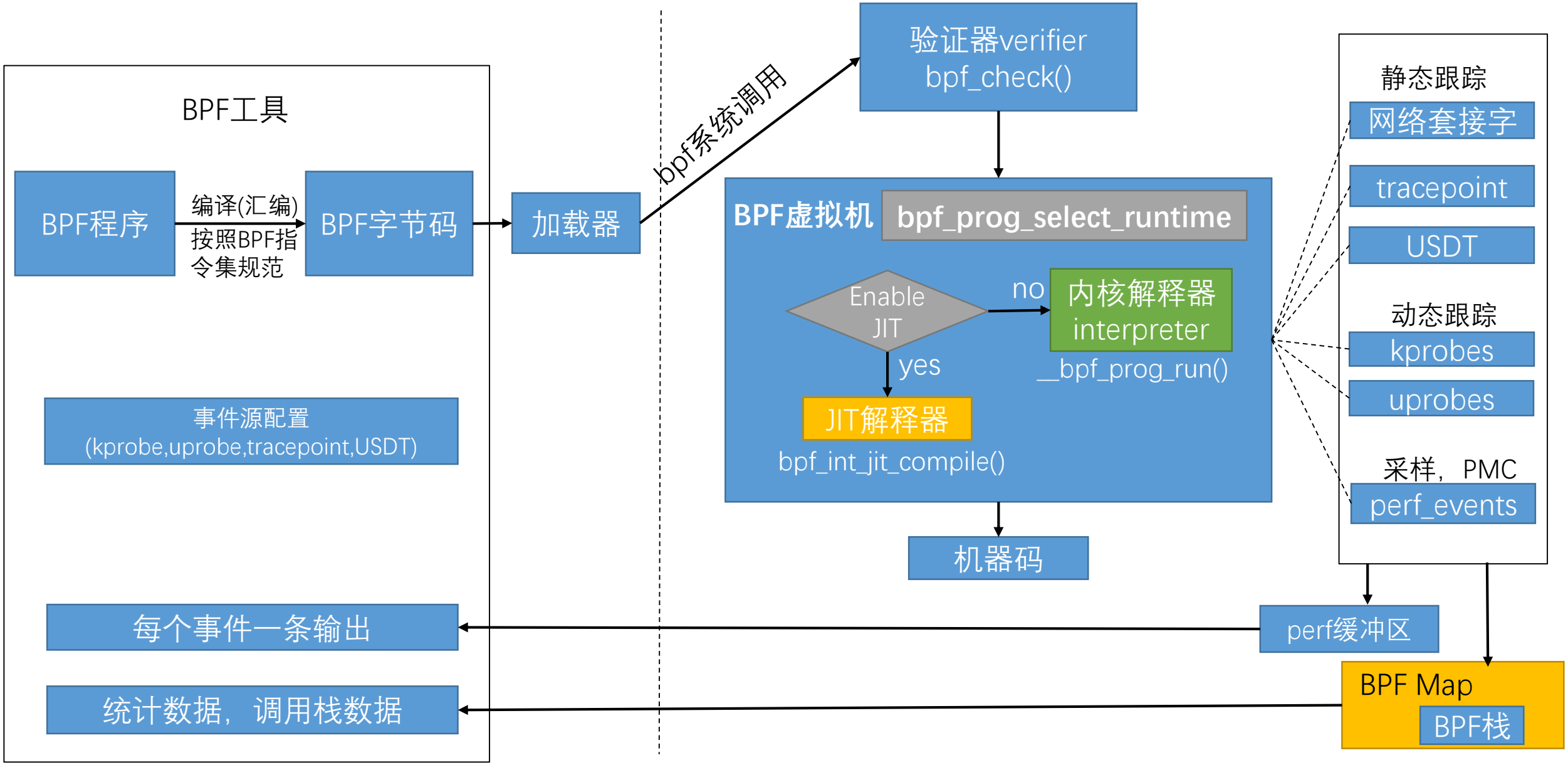
```c
unsigned int sk_run_filter(struct sk_buff *skb, \
                           struct sock_filter *filter, int flen)
{
    struct sock_filter *fentry; /* We walk down these */
    void *ptr;
    u32 A = 0;              /* Accumulator */
    u32 X = 0;              /* Index Register */
    u32 mem[BPF_MEMWORDS];      /* Scratch Memory Store */
    u32 tmp;
    int k;
    int pc;

    /*
     * Process array of filter instructions.
     */
    for (pc = 0; pc < flen; pc++) {
        fentry = &filter[pc];
        ...
        switch (fentry->code) {
        case BPF_ALU|BPF_ADD|BPF_X:
            A += X;
            continue;
        case BPF_ALU|BPF_SUB|BPF_X:
            A -= X;
            continue;
        case BPF_ALU|BPF_SUB|BPF_K:
            A -= fentry->k;
            continue;
        case BPF_ALU|BPF_MUL|BPF_X:
            A *= X;
            continue;
        ...
    }
    return 0;
}
```

报文到 —触发→ packet_rcv() —call→

事件　　注册的回调函数

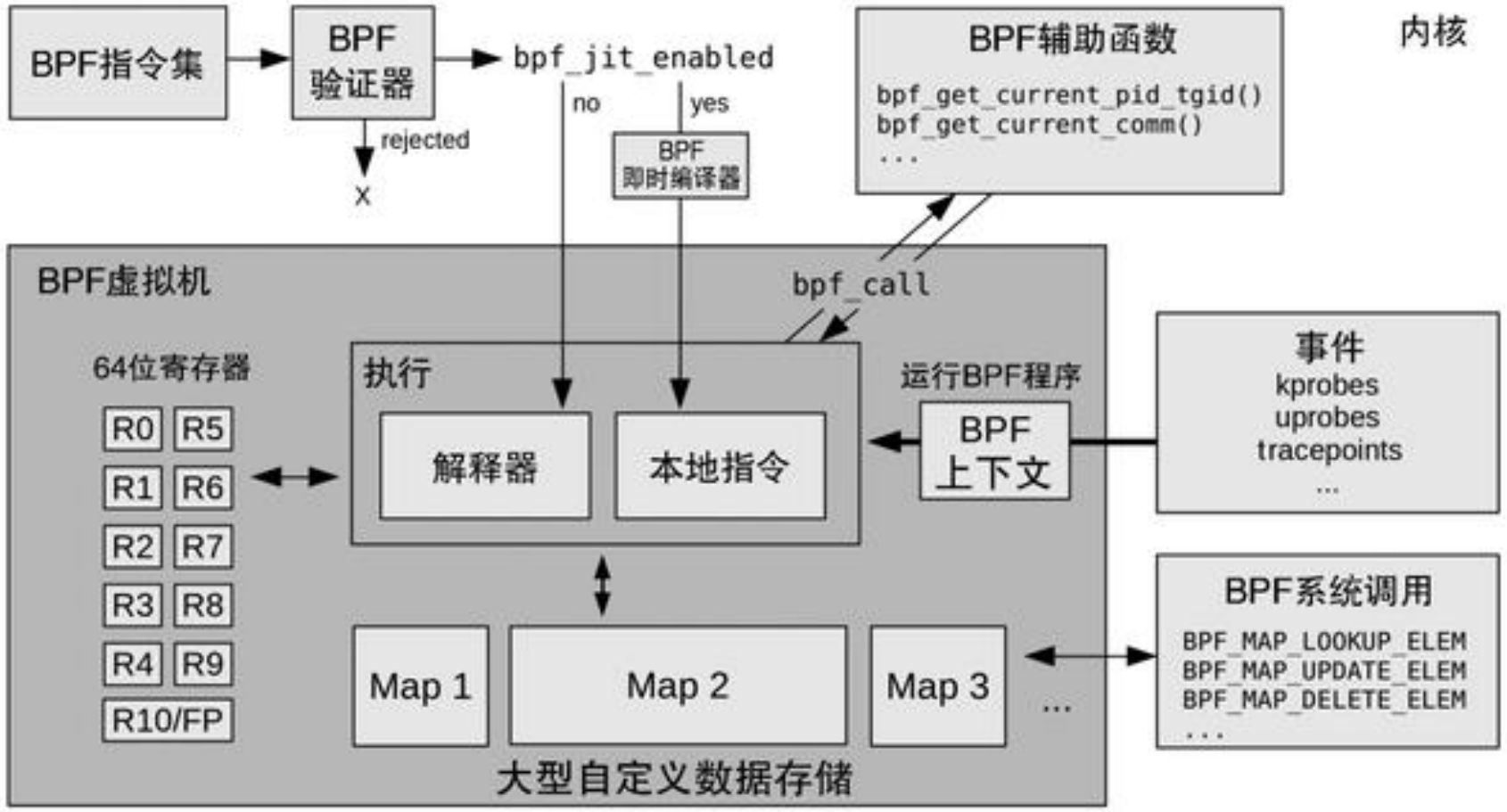

eBPF

扩展的BPF机制(eBPF)技术架构图



BPF运行时结构

Linux BPF运行时各模块示意图：



BPF虚拟机的实现既包括一个解释器，又包括一个JIT编译器：JIT编译器负责生成处理器可直接执行的机器指令

BPFvs eBPF：扩展的BPF扩展了什么？

1. eBPF的常规项扩展

对比项	经典BPF	扩展版BPF
寄存器数量	2个：寄存器A和寄存器X	10个：R0~R9，此外R10是只读的帧指针寄存器
寄存器宽度	32位	64位
存储	16个内存槽位：M[0-15]	512字节大小的栈空间，外加无限制的映射型存储Map
受限的内核调用	非常受限，JIT专用	可用，通过BPF_CALL指令
支持的事件类型	网络数据包，seccomp-BPF	网络数据包，内核函数，用户态函数，跟踪点，用户态标记，PMC

在x86_64上，所有寄存器都一一映射到硬件寄存器。例如，x86_64的JIT编译器可以将它们映射为：

R0 - rax
R1 - rdi
R2 - rsi
R3 - rdx
R4 - rcx
R5 - r8
R6 - rbx
R7 - r13
R8 - r14
R9 - r15
R10 – rbp

BPFvs eBPF：扩展的BPF扩展了什么？

2. 指令集扩展

名称	类型	来源	编号	描述
BPF_LD	指令类	经典版	0x00	加载
BPF_LDX	指令类	经典版	0x01	加载到X
BPF_ST	指令类	经典版	0x02	存储
BPF_STX	指令类	经典版	0x03	存储到X
BPF_JMP	指令类	经典版	0x05	跳转
BPF_RET	指令类	经典版	0x06	返回
…	…	…	…	…
BPF_ALU64	指令类	扩展版	0x07	ALU64位
BPF_DW	大小	扩展版	0x18	64位双字
BPF_MOV	ALU/跳跃操作	扩展版	0xb0	寄存器间移动
BPF_K	ALU/跳跃操作	扩展版	0x00	现值操作符
BPF_REG_1	寄存器编号	扩展版	0x01	1号寄存器
…	…	…	…	…

缩写解释:
LD：加载
LDX：从寄存器加载
ST：存储
STX：存储到寄存器
JMP：跳转
MOV：移动

BPF的基础设施

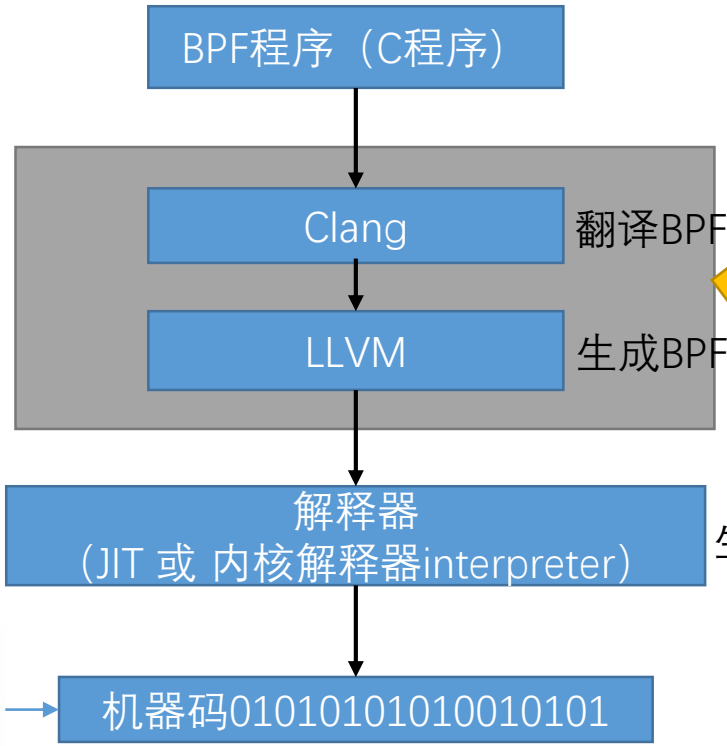
BPF 不仅仅是一个指令集，它还提供了围绕自身的一些基础设施，例如：

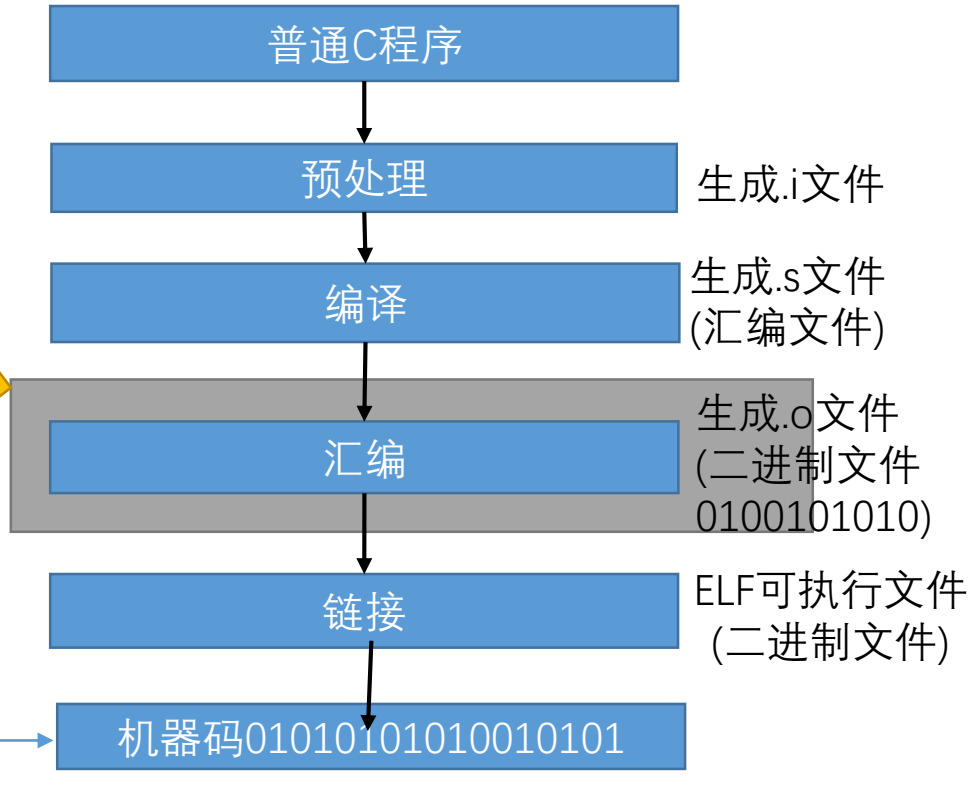
- BPF存储对象： BPF map
- BPF辅助函数（helper function）： 可以更方便地利用内核功能或与内核交互
- 尾调用（tail call）： 高效地调用其他 BPF 程序
- 安全加固原语（security hardening primitives）： bpf_spin_lock(lock)、bpf_spin_unlock(lock)
- 用于钉住（pin）对象（例如 map、程序）的伪文件系统： BPF sysfs(sys/fs/bpf)
- LLVM 提供了一个 BPF 后端（back end）： 因此使用 clang 这样的工具就可以将 C 代 码编译成 BPF 对象文件（object file）， 最后生成BPF字节码

如何理解BPF的指令集？

BPF指令集区别于通用的X86和ARM指令集。

- BPF指令集采用虚拟指令集规范。BPF指令集中的指令类似于汇编，如汇编的无条件跳转指令为jmp，而BPF指令集中则为BPF_JMP。
- X86和ARM指令集，每一条指令对应的是一条特定的逻辑门电路。

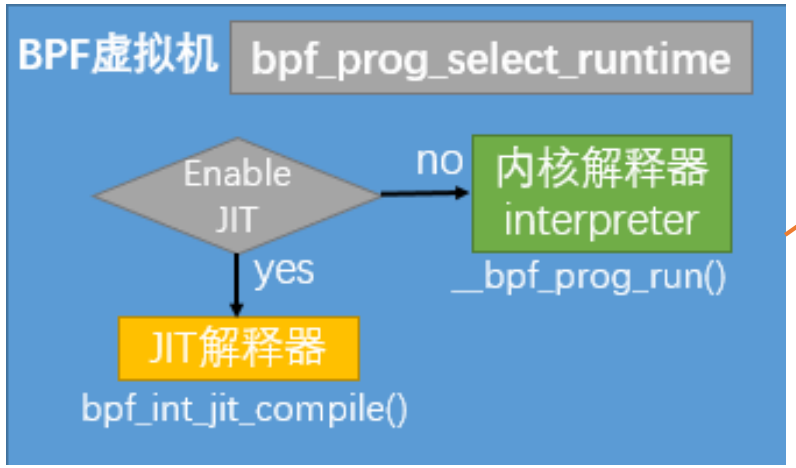




BPF虚拟机与BPF指令集

了解BPF指令集有什么用?

- 更好的理解BPF虚拟机如何工作



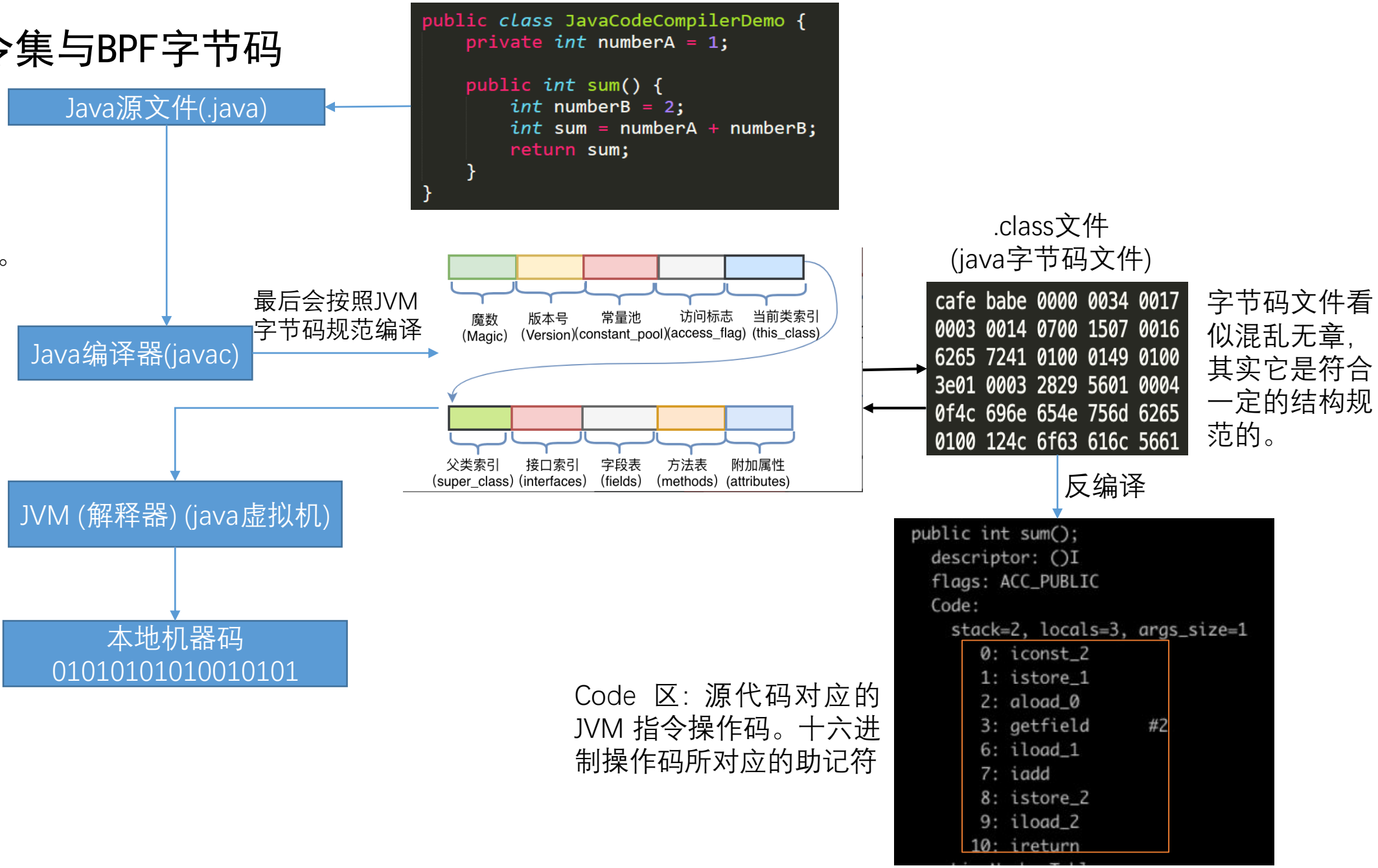
BPF虚拟机可看作解释器。在eBPF机制的架构图里，BPF虚拟机做的就是解释器的工作。

```
/**
 * Decode and execute eBPF instructions.
 */
static unsigned int __bpf_prog_run(void *ctx, const struct bpf_insn *insn)
{
        ...
        /* 64 bit ALU operations */
        [BPF_ALU64 | BPF_ADD | BPF_X] = &&ALU64_ADD_X,
        [BPF_ALU64 | BPF_ADD | BPF_K] = &&ALU64_ADD_K,
        [BPF_ALU64 | BPF_SUB | BPF_X] = &&ALU64_SUB_X,
        [BPF_ALU64 | BPF_SUB | BPF_K] = &&ALU64_SUB_K,
        [BPF_ALU64 | BPF_AND | BPF_X] = &&ALU64_AND_X,
        [BPF_ALU64 | BPF_AND | BPF_K] = &&ALU64_AND_K,
        [BPF_ALU64 | BPF_OR | BPF_X] = &&ALU64_OR_X,
        [BPF_ALU64 | BPF_OR | BPF_K] = &&ALU64_OR_K,
        [BPF_ALU64 | BPF_LSH | BPF_X] = &&ALU64_LSH_X,
        [BPF_ALU64 | BPF_LSH | BPF_K] = &&ALU64_LSH_K,
        [BPF_ALU64 | BPF_RSH | BPF_X] = &&ALU64_RSH_X,
        [BPF_ALU64 | BPF_RSH | BPF_K] = &&ALU64_RSH_K,
        [BPF_ALU64 | BPF_XOR | BPF_X] = &&ALU64_XOR_X,
        [BPF_ALU64 | BPF_XOR | BPF_K] = &&ALU64_XOR_K,
        [BPF_ALU64 | BPF_MUL | BPF_X] = &&ALU64_MUL_X,
        [BPF_ALU64 | BPF_MUL | BPF_K] = &&ALU64_MUL_K,
        [BPF_ALU64 | BPF_MOV | BPF_X] = &&ALU64_MOV_X,
        [BPF_ALU64 | BPF_MOV | BPF_K] = &&ALU64_MOV_K,
        [BPF_ALU64 | BPF_ARSH | BPF_X] = &&ALU64_ARSH_X,
        [BPF_ALU64 | BPF_ARSH | BPF_K] = &&ALU64_ARSH_K,
        [BPF_ALU64 | BPF_DIV | BPF_X] = &&ALU64_DIV_X,
        [BPF_ALU64 | BPF_DIV | BPF_K] = &&ALU64_DIV_K,
        [BPF_ALU64 | BPF_MOD | BPF_X] = &&ALU64_MOD_X,
        [BPF_ALU64 | BPF_MOD | BPF_K] = &&ALU64_MOD_K,
        [BPF_ALU64 | BPF_NEG] = &&ALU64_NEG,
        ...
    };
```

BPF指令集与BPF字节码

什么是字节码?

- 字节码也是一种可以被执行的“机器码”，只不过被虚拟机执行。之所以称之为字节码，是指这里面的操作码(opcode)是一个字节长。一般机器指令由操作码和操作数组成，字节码(虚拟的机器码)也是由操作码(opcode)和操作数(op)组成。对于字节码，它是按照一套虚机指令集格式来组织。



Code 区：源代码对应的 JVM 指令操作码。十六进制操作码所对应的助记符

BPF指令集与BPF字节码

BPF字节码

```
# bpftrace -v biolatency.bt
Attaching 4 probes...

Program ID: 677

Bytecode:
0: (bf) r6 = r1
1: (b7) r1 = 29810
2: (6b) *(u16 *)(r10 -4) = r1
3: (b7) r1 = 1635021632
4: (63) *(u32 *)(r10 -8) = r1
5: (b7) r1 = 20002
6: (7b) *(u64 *)(r10 -16) = r1
7: (b7) r1 = 0
8: (73) *(u8 *)(r10 -2) = r1
9: (18) r7 = 0xffff96e697298800
11: (85) call bpf_get_smp_processor_id#8
12: (bf) r4 = r10
13: (07) r4 += -16
14: (bf) r1 = r6
15: (bf) r2 = r7
16: (bf) r3 = r0
17: (b7) r5 = 15
18: (85) call bpf_perf_event_output#25
19: (b7) r0 = 0
20: (95) exit
[...]
```

用类似于汇编指令的方式让人类能理解这个字节码的含义

BPF指令集与BPF字节码

由BPF指令如何变为BPF字节码?

```
int main(int argc, char *argv[])
{
    struct bpf_insn prog[] = {
        BPF_MOV64_IMM(BPF_REG_1, 0xa21),
        BPF_STX_MEM(BPF_H, BPF_REG_10, BPF_REG_1, -4),
        BPF_MOV64_IMM(BPF_REG_1, 0x646c726f),
        ...
        BPF_EXIT_INSN(),
    };
    ...
    return 0;
}
```

使用BPF指令宏将BPF程序声明为prog数组

名称	编号
BPF_ALU64	0X07
BPF_MOV	0xb0
BPF_K	0x00

BPF_MOV64_IMM (BPF_REG_1，0xa21)

BPF指令辅助宏（将64位的立即数移动到目的地）

展开

BPF_ALU64 | BPF_MOV | BPF_K　BPF指令

(opcode) 操作码　(opcode) 操作码　(opcode) 操作码

0xb7　0x01　0xa21　eBPF指令格式的元数据

编译器

操作码	目标寄存器	源寄存器	有符号偏移量	有符号立即常量
8位	8位	8位	16位	32位

按照BPF指令规范翻译生成字节码

b7 01 00 00 21 0a 00 00　BPF字节码

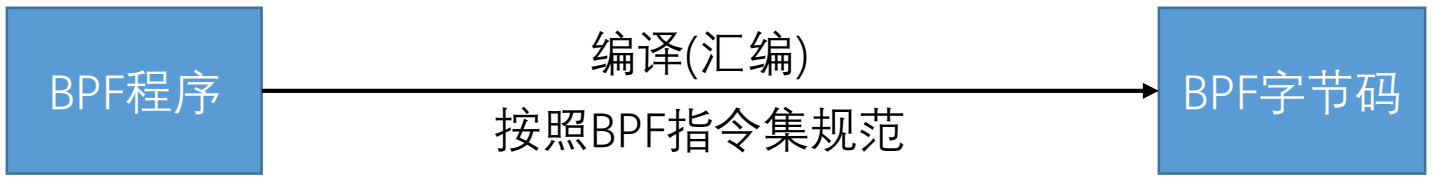
扩展的BPF（eBPF）指令格式

eBPF指令格式：

操作码	目标寄存器	源寄存器	有符号偏移量	有符号立即常量
8位	8位	8位	16位	32位

eBPF指令格式的定义：

```
struct bpf_insn {
    __u8    code;           /* opcode */
    __u8    dst_reg:4;      /* dest register */
    __u8    src_reg:4;      /* source register */
    __s16   off;            /* signed offset */
    __s32   imm;            /* signed immediate constant */
};
```



BPF虚拟机（内核解释器）

BPF虚拟机解码(decode)并执行(execute)

BPF_ALU64 | BPF_MOV | BPF_K = 0xb7

```
/**
 * Decode and execute eBPF instructions.
 */
static unsigned int __bpf_prog_run(void *ctx, const struct bpf_insn *insn)
{
    ...
    static const void *jumptable[256] = {
        ...
        [BPF_ALU64 | BPF_MOV | BPF_K] = &&ALU64_MOV_K,
        ...
    }

#undef ALU
    ...
    ALU64_MOV_K:
        DST = IMM;
        CONT;
    ...
        return 0;
}
```

DST = IMM

目的地 (目标寄存器)

立即数 (常量)

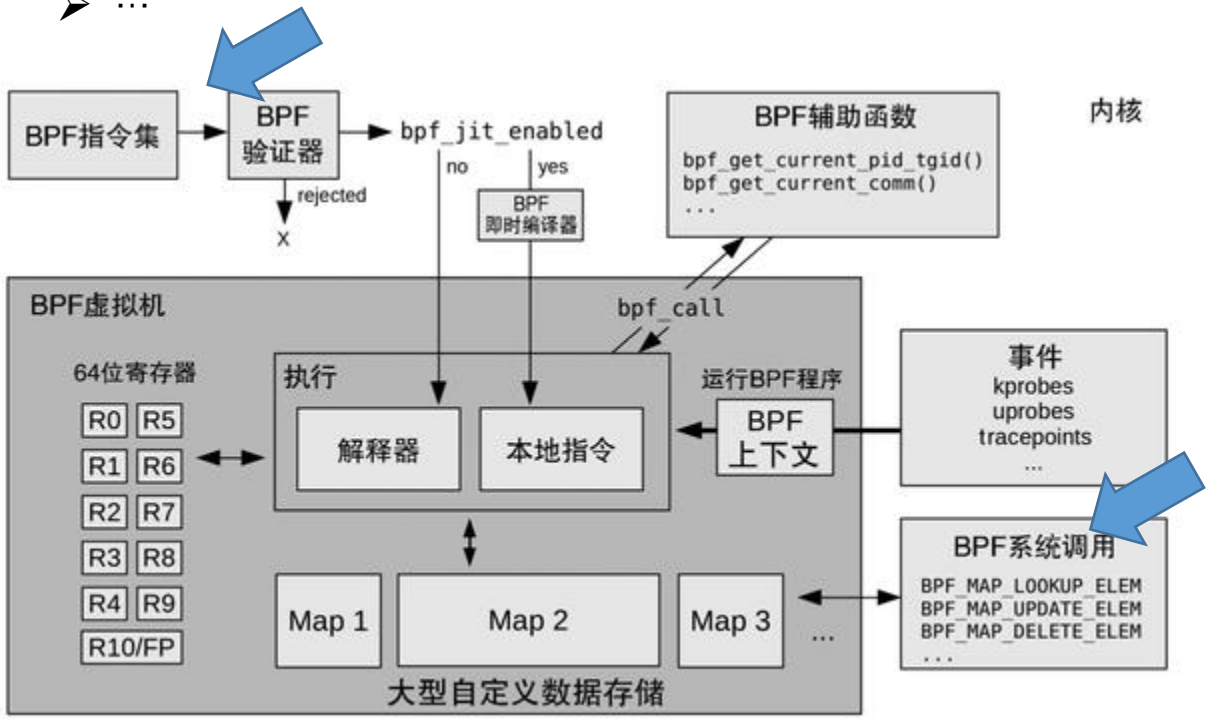
```
# bpftrace -v biolatency.bt
Attaching 4 probes...

Program ID: 677

Bytecode:
0: (bf) r6 = r1
1: (b7) r1 = 29810
2: (6b) *(u16 *)(r10 -4) = r1
3: (b7) r1 = 1635021632
4: (63) *(u32 *)(r10 -8) = r1
5: (b7) r1 = 20002
6: (7b) *(u64 *)(r10 -16) = r1
7: (b7) r1 = 0
8: (73) *(u8 *)(r10 -2) = r1
9: (18) r7 = 0xffff96e697298800
11: (85) call bpf_get_smp_processor_id#8
12: (bf) r4 = r10
13: (07) r4 += -16
14: (bf) r1 = r6
15: (bf) r2 = r7
16: (bf) r3 = r0
17: (b7) r5 = 15
18: (85) call bpf_perf_event_output#25
19: (b7) r0 = 0
20: (95) exit
[...]
```

BPF程序的编译、加载和运行

BPF系统调用：bpf()

bpf()系统调用的作用

- 加载BPF程序
- 操作Map（增、删、改、查）
- …



Kernel/bpf/syscall.c

```
SYSCALL_DEFINE3(bpf, int, cmd, union bpf_attr __user *, uattr, unsigned int, size)
{
    ...
    switch (cmd) {
    case BPF_MAP_CREATE:
        err = map_create(&attr);
        break;
    case BPF_MAP_LOOKUP_ELEM:
        err = map_lookup_elem(&attr);
        break;
    case BPF_MAP_UPDATE_ELEM:
        err = map_update_elem(&attr);
        break;
    case BPF_MAP_DELETE_ELEM:
        err = map_delete_elem(&attr);
        break;
    case BPF_MAP_GET_NEXT_KEY:
        err = map_get_next_key(&attr);
        break;
    case BPF_PROG_LOAD:
        err = bpf_prog_load(&attr);
        break;
    case BPF_OBJ_PIN:
        err = bpf_obj_pin(&attr);
        break;
    case BPF_OBJ_GET:
        err = bpf_obj_get(&attr);
        break;
    case BPF_PROG_ATTACH:
        err = bpf_prog_attach(&attr);
        break;
    case BPF_PROG_DETACH:
        err = bpf_prog_detach(&attr);
        break;
    ...
    case BPF_MAP_LOOKUP_AND_DELETE_ELEM:
        err = map_lookup_and_delete_elem(&attr);
        break;
    ...
    }
    return err;
}
```

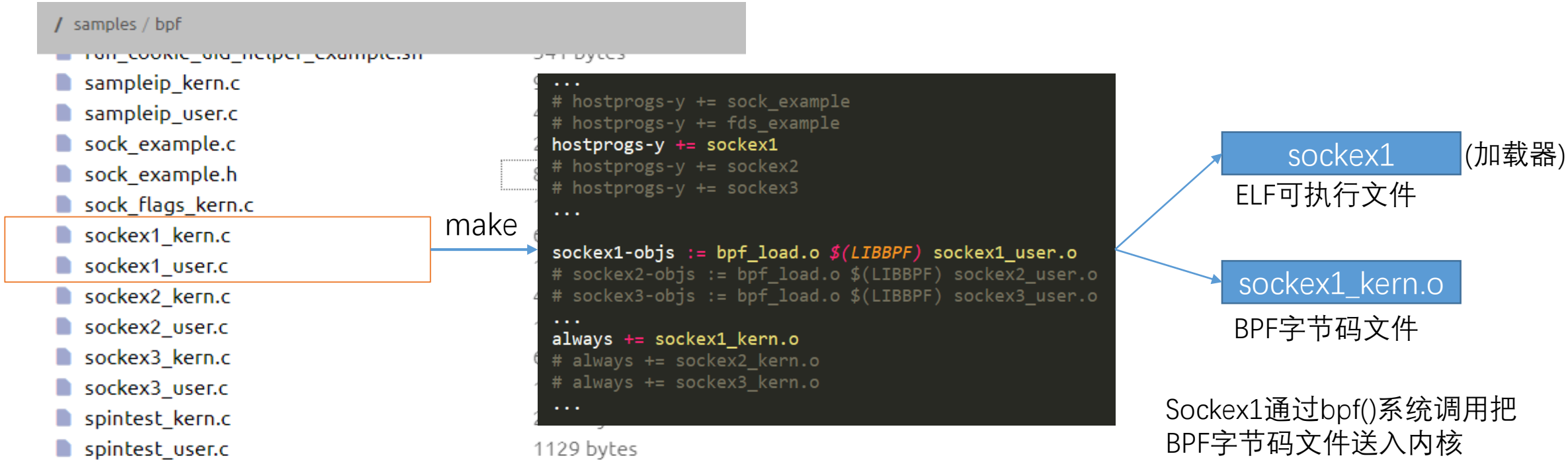
bpf描述符(bpf_prog)、BPF程序类型、BPF存储类型

每一个 load 到内核的 eBPF 程序都有一个 fd 会返回给用户，它对应一个 bpf_prog结构。

```
struct bpf_prog {
    u16             pages;              /* Number of allocated pages */
    u16             jited:1,            /* Is our filter JIT'ed? */
                    jit_requested:1,     /* archs need to JIT the prog */
                    undo_set_mem:1,      /* Passed set_memory_ro() checkpoint */
                    gpl_compatible:1,    /* Is filter GPL compatible? */
                    cb_access:1,         /* Is control block accessed? */
                    dst_needed:1,        /* Do we need dst entry? */
                    blinded:1,          /* Was blinded */
                    is_func:1,          /* program is a bpf function */
                    kprobe_override:1,   /* Do we override a kprobe? */
                    has_callchain_buf:1;  /* callchain buffer allocated? */
    enum bpf_prog_type  type;           /* Type of BPF program */ //当前bpf程序的类型(kprobe/tracepoint/perf_event...)
    enum bpf_attach_type    expected_attach_type; /* For some prog types */ // 程序包含 bpf 指令的数量
    u32             len;                /* Number of filter blocks */
    u32             jited_len;          /* Size of jited insns in bytes */
    u8              tag[BPF_TAG_SIZE];
    struct bpf_prog_aux *aux;               /* Auxiliary fields */ // 主要用来辅助 verifier 校验和转换的数据
    struct sock_fprog_kern  *orig_prog;         /* Original BPF program */
    unsigned int        (*bpf_func)(const void *ctx, const struct bpf_insn *insn); /* Instructions for interpreter */
                                                // 运行时BPF程序的入口。如果JIT转换成功，这里指向的就是BPF程序JIT转换后的映像；
                                                // 否则这里指向内核解析器(interpreter)的通用入口__bpf_prog_run()
    union {
        struct sock_filter  insns[0];
        struct bpf_insn     insnsi[0];    // 从用户态拷贝过来的，BPF程序原始指令的存放空间
    };
};
```

用户空间的BPF程序如何载入内核

- 在 eBPF 中，用户可以用 C 语言编写最后需要在内核空间运行的代码，clang 编译器会将需要灌入到内核的代码编译成 .o 文件(BPF字节码包含于其中)，之后用户可以通过编写用户空间程序，载入 .o 文件，完成内核空间程序的注入。
- eBPF 既支持在内核源码树编译，也支持使用 bcc 脱离源码树编译。以使用内核源码树编译为例。
- 在内核代码的 samples/bpf 目录，有很多现成的例子，这里选择 sockex1。在此目录下执行 make 就可以编译所有例子。



用户空间的BPF程序如何载入内核

统计各个协议报文的数据量

函数 bpf_prog1 被 SEC("socket1") 修饰，它表示该函数会被编译到名为 socket1 的 section中.

```
#define SEC(NAME) __attribute__((section(NAME), used))
```

sockex1_kern.c

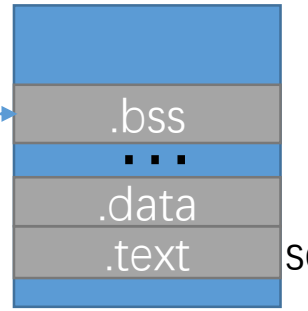
```c
#include <uapi/linux/if_ether.h>
#include <uapi/linux/if_packet.h>
#include <uapi/linux/ip.h>
#include "bpf_helpers.h"

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(long),
    .max_entries = 256,
};

SEC("socket1")
int bpf_prog1(struct __sk_buff *skb)
{
    int index = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
    long *value;

    if (skb->pkt_type != PACKET_OUTGOING)
        return 0;

    value = bpf_map_lookup_elem(&my_map, &index);
    if (value)
        __sync_fetch_and_add(value, skb->len);

    return 0;
}
char _license[] SEC("license") = "GPL";
```

sockex1_user.c

```c
int main(int ac, char **argv)
{
    char filename[256];
    FILE *f;
    int i, sock;

    snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);

    /* 装载文件 sockex1_kern.o */
    if (load_bpf_file(filename)) {
        printf("%s", bpf_log_buf);
        return 1;
    }
    ...
    /* 循环读取 map_fd[0] 对应存储区域的各个协议类型对应的统计计数并显示 */
    for (i = 0; i < 5; i++) {
        long long tcp_cnt, udp_cnt, icmp_cnt;
        int key;

        key = IPPROTO_TCP;
        assert(bpf_map_lookup_elem(map_fd[0], &key, &tcp_cnt) == 0);
        ...
        sleep(1);
    }

    return 0;
}
```

编译

生成



section(节)

ELF可执行文件

Makefile → clang+LLVM编译

sockex1_kern.o

BPF字节码文件

通过bpf()系统调用将字节码文件注入到内核中

Linux内核

用户空间的BPF程序如何载入内核

扫描时关心的 section 的名字的前缀

```
load_bpf_file
  |
  |- do_load_bpf_file

int do_load_bpf_file(const char *path, fixup_map_cb fixup_map)
{
    fd = open(path, O_RDONLY, 0);
    ......
    /* load programs */
    for (i = 1; i < ehdr.e_shnum; i++) {
        ......
        if (memcmp(shname, "kprobe/", 7) == 0 ||
            memcmp(shname, "kretprobe/", 10) == 0 ||
            memcmp(shname, "tracepoint/", 11) == 0 ||
            memcmp(shname, "xdp", 3) == 0 ||
            memcmp(shname, "perf_event", 10) == 0 ||
            memcmp(shname, "socket", 6) == 0 ||
            memcmp(shname, "cgroup/", 7) == 0 ||
            memcmp(shname, "sockops", 7) == 0 ||
            memcmp(shname, "sk_skb", 6) == 0) {
            ret = load_and_attach(shname, data->d_buf,
                                  data->d_size);
            if (ret != 0)
                goto done;
        }
    }
}
```

do_load_bpf_file 会将输入的 .o 文件作为 ELF 格式文件的逐个 section 进行分析，如 section 的名字是特殊的(比如“socket”)，那么就会将这个section 的内容作为 load_and_attach() 的参数。

```
static int load_and_attach(const char *event, struct bpf_insn *prog, int size)
{
    bool is_socket = strncmp(event, "socket", 6) == 0;
    ......

    if (is_socket) {
        prog_type = BPF_PROG_TYPE_SOCKET_FILTER;
    }
    ......

    fd = bpf_load_program(prog_type, prog, insns_cnt, license, kern_version,
                          bpf_log_buf, BPF_LOG_BUF_SIZE);
```

```
bpf_load_program
  |
  |-- bpf_load_program_name

int bpf_load_program_name(enum bpf_prog_type type, const char *name,
              const struct bpf_insn *insns,
              size_t insns_cnt, const char *license,
              __u32 kern_version, char *log_buf,
              size_t log_buf_sz)
{
    int fd;
    union bpf_attr attr;
    __u32 name_len = name ? strlen(name) : 0;
    ...
    attr.prog_type = type;
    attr.insn_cnt = (__u32)insns_cnt;
    attr.insns = ptr_to_u64(insns);
    attr.license = ptr_to_u64(license);
    ...
    attr.kern_version = kern_version;
    memcpy(attr.prog_name, name, min(name_len, BPF_OBJ_NAME_LEN - 1));

    fd = sys_bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
    if (fd >= 0 || !log_buf || !log_buf_sz)
        return fd;
    ......
}
```

用户空间的BPF程序如何载入内核

```c
SYSCALL_DEFINE3(bpf, int, cmd, union bpf_attr __user *, uattr, unsigned int, size)
{
    ......
    case BPF_PROG_LOAD:
        err = bpf_prog_load(&attr);
}

static int bpf_prog_load(union bpf_attr *attr)
{
    struct bpf_prog *prog;

    ......
    /* 分配内核 bpf_prog 程序数据结构空间 */
    prog = bpf_prog_alloc(bpf_prog_size(attr->insn_cnt), GFP_USER);
    ......
    /* 将 bpf 虚拟机指令从用户空间拷贝到内核空间 */
    copy_from_user(prog->insns, u64_to_user_ptr(attr->insns), bpf_prog_insn_size(prog));
    ......
    /* run eBPF verifier */
    // 使用 verifer 对 BPF 程序进行合法性扫描
    err = bpf_check(&prog, attr);

    /* 分配一个 fd 与 prog 关联, 最终这个 fd 将返回用户空间 */
    err = bpf_prog_new_fd(prog);
    ......

    return err;
}
```

```c
#ifndef SYSCALL_DEFINE0
#define SYSCALL_DEFINE0(sname)                  \
    SYSCALL_METADATA(_##sname, 0);              \
    asmlinkage long sys_##sname(void);          \
    ALLOW_ERROR_INJECTION(sys_##sname, ERRNO);      \
    asmlinkage long sys_##sname(void)
#endif /* SYSCALL_DEFINE0 */

#define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE4(name, ...) SYSCALL_DEFINEx(4, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, _##name, __VA_ARGS__)
#define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, _##name, __VA_ARGS__)

#define SYSCALL_DEFINE_MAXARGS  6

#define SYSCALL_DEFINEx(x, sname, ...)              \
    SYSCALL_METADATA(sname, x, __VA_ARGS__)         \
    __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

内核空间 load BPF 指令

```c
asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

BPF虚机机执行的时机

BPF程序指令何时执行?

- eBPF 程序指令都是在内核的特定 Hook 点执行，不同类型的程序有不同的钩子，有不同的上下文(ctx)

将指令 load 到内核时，内核会创建 bpf_prog 存储指令，但只是第一步，成功运行这些指令还需要完成以下两个步骤:

- 将 bpf_prog 与内核中的特定 Hook 点关联起来，也就是将BPF程序挂到钩子上。
- 在 Hook 点被访问到时，取出 bpf_prog，执行这些指令。

(举例：若某个kprobe探测点的内核地址attach了一段BPF程序后，当内核执行到这个地址时发生陷入(trap)，唤醒kprobe的回调函数，后者又会触发attach的BPF程序执行。

BPF程序类型（Hook点）

/include/uapi/linux/bpf.h

```
enum bpf_prog_type {
    BPF_PROG_TYPE_UNSPEC,
    BPF_PROG_TYPE_SOCKET_FILTER,
    BPF_PROG_TYPE_KPROBE,
    BPF_PROG_TYPE_SCHED_CLS,
    BPF_PROG_TYPE_SCHED_ACT,
    BPF_PROG_TYPE_TRACEPOINT,
    BPF_PROG_TYPE_XDP,
    BPF_PROG_TYPE_PERF_EVENT,
    BPF_PROG_TYPE_CGROUP_SKB,
    BPF_PROG_TYPE_CGROUP_SOCK,
    BPF_PROG_TYPE_LWT_IN,
    BPF_PROG_TYPE_LWT_OUT,
    BPF_PROG_TYPE_LWT_XMIT,
    BPF_PROG_TYPE_SOCK_OPS,
    BPF_PROG_TYPE_SK_SKB,
    BPF_PROG_TYPE_CGROUP_DEVICE,
};
```

BPF程序类型介绍

bpf_prog_type	描述
BPF_PROG_TYPE_KPROBE	用于内核动态插桩点kprobes
BPF_PROG_TYPE_TRACEPOINT	用于内核静态跟踪点
BPF_PROG_TYPE_PERF_EVENT	用于perf_events，包括PMC
…	…
BPF_PROG_TYPE_SOCKET_FILTER	用于挂载到网络套接字上(最早的BPF使用场景)
BPF_PROG_TYPE_SCHED_CLS	用于流量控制分类
BPF_PROG_TYPE_XDP	用于XDP(eXpress Data Path)程序
…	…

BPF存储类型(Map)

/include/uapi/linux/bpf.h

```
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC,
    BPF_MAP_TYPE_HASH,
    BPF_MAP_TYPE_ARRAY,
    BPF_MAP_TYPE_PROG_ARRAY,
    BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    BPF_MAP_TYPE_PERCPU_HASH,
    BPF_MAP_TYPE_PERCPU_ARRAY,
    BPF_MAP_TYPE_STACK_TRACE,
    BPF_MAP_TYPE_CGROUP_ARRAY,
    BPF_MAP_TYPE_LRU_HASH,
    BPF_MAP_TYPE_LRU_PERCPU_HASH,
    BPF_MAP_TYPE_LPM_TRIE,
    BPF_MAP_TYPE_ARRAY_OF_MAPS,
    BPF_MAP_TYPE_HASH_OF_MAPS,
    BPF_MAP_TYPE_DEVMAP,
    BPF_MAP_TYPE_SOCKMAP,
    BPF_MAP_TYPE_CPUMAP,
    BPF_MAP_TYPE_XSKMAP,
    BPF_MAP_TYPE_SOCKHASH,
    BPF_MAP_TYPE_CGROUP_STORAGE,
    BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
};
```

BPF存储类型（Map）

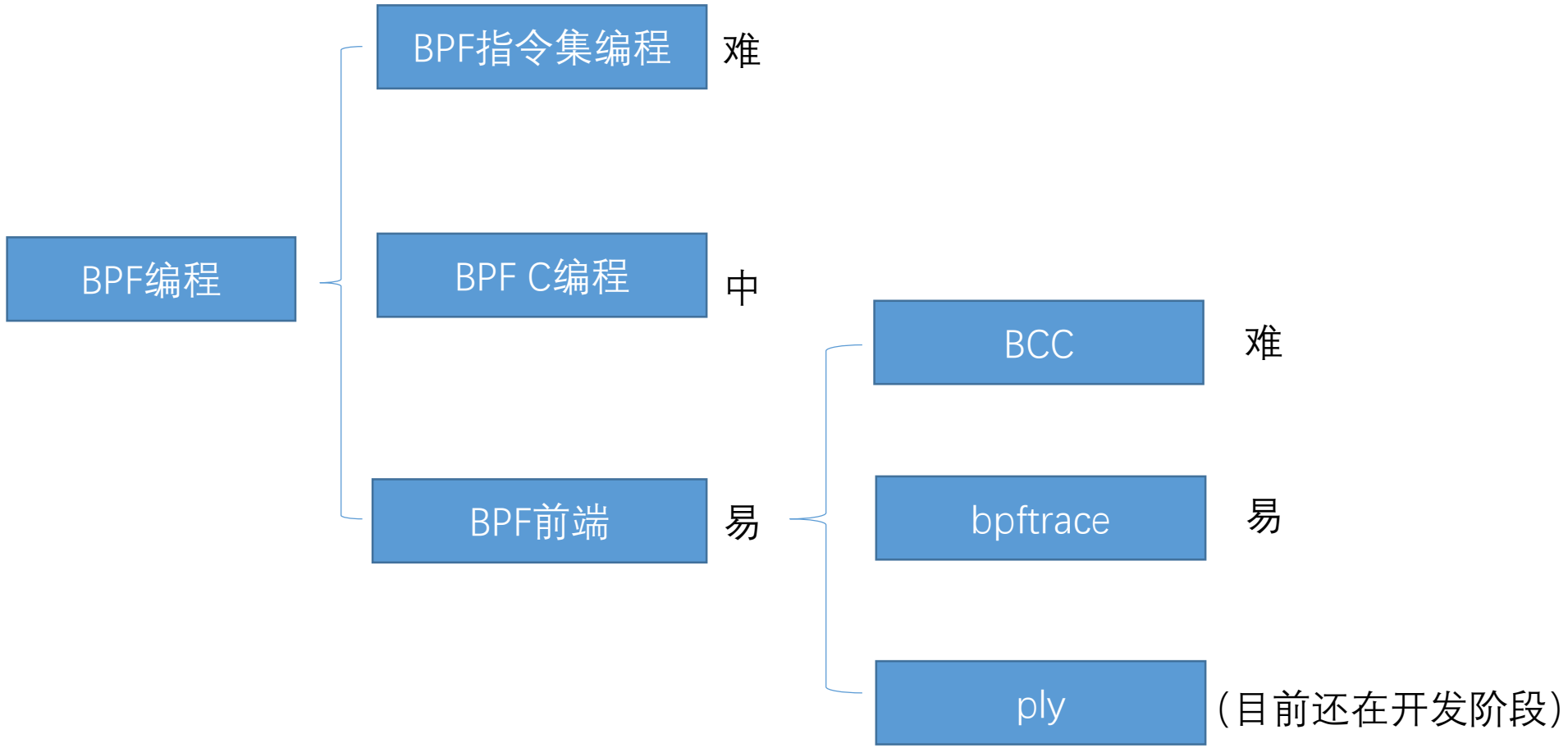
bpf_map_type	描述
BPF_MAP_TYPE_HASH	用于哈希表的Map类型：保存key/value对
BPF_MAP_TYPE_ARRAY	数组类型
BPF_MAP_TYPE_PERF_EVENT_ARRAY	到perf_event环形缓冲区的接口，用于将记录发送到用户空间
BPF_MAP_TYPE_PERCPU_HASH	一个基于每CPU单独维护的更快哈希表
BPF_MAP_TYPE_PERCPU_ARRAY	一个基于每CPU单独维护的更快数组
BPF_MAP_TYPE_STACK_TRACE	调用栈存储，使用栈ID进行索引
BPF_MAP_TYPE_STACK	调用栈存储
…	…

BPF辅助函数

BPF程序不能随意调用内核函数，内核专门提供了BPF可以调用的函数

BPF辅助函数	描述
bpf_map_lookup_elem(map, key)	在map中查找键值key，并返回它的值(指针)
bpf_map_delete_elem(map, key)	根据key值删除map中对应的元素
bpf_ktime_get_ns()	返回系统启动后的时长，单位ns
bpf_trace_printk(fmt, fmt_size, …)	向TraceFS的pipe文件中写入调试信息
…	…

如何使用BPF：BPF编程



BPF指令集编程

```c
#include <stdio.h>
#include <linux/version.h>
#include <bpf/bpf.h>
...

int main(int argc, char *argv[])
{
    struct bpf_insn prog[] = {
        BPF_MOV64_IMM(BPF_REG_1, 0xa21),
        BPF_STX_MEM(BPF_H, BPF_REG_10, BPF_REG_1, -4),
        BPF_MOV64_IMM(BPF_REG_1, 0x646c726f),
        ...
        BPF_EXIT_INSN(),
    };
    ...

    size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);

    int prog_fd = bpf_load_program(BPF_PROG_TYPE_KPROBE, prog, insns_cnt,
                                   "GPL", LINUX_VERSION_CODE,
                                   bpf_log_buf, BPF_LOG_BUF_SIZE);
    ...
    int probe_fd = bpf_attach_kprobe(prog_fd, BPF_PROBE_ENTRY, "hello_world",
                                   "do_nanosleep", 0, 0);
    ...
    bpf_detach_kprobe("hello_world");

    return 0;
}
```

BPF C编程

编写kern.c+编写user.c+修改Makefile

sockex1_kern.c

```c
#include <uapi/linux/if_ether.h>
#include <uapi/linux/if_packet.h>
#include <uapi/linux/ip.h>
#include "bpf_helpers.h"

struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(long),
    .max_entries = 256,
};

SEC("socket1")
int bpf_prog1(struct __sk_buff *skb)
{
    int index = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
    long *value;

    if (skb->pkt_type != PACKET_OUTGOING)
        return 0;

    value = bpf_map_lookup_elem(&my_map, &index);
    if (value)
        __sync_fetch_and_add(value, skb->len);

    return 0;
}
char _license[] SEC("license") = "GPL";
```

sockex1_user.c

```c
int main(int ac, char **argv)
{
    char filename[256];
    FILE *f;
    int i, sock;

    snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);

    /* 装载文件 sockex1_kern.o  */
    if (load_bpf_file(filename)) {
        printf("%s", bpf_log_buf);
        return 1;
    }
    ...
    /* 循环读取 map_fd[0] 对应存储区域的各个协议类型对应的统计计数并显示 */
    for (i = 0; i < 5; i++) {
        long long tcp_cnt, udp_cnt, icmp_cnt;
        int key;

        key = IPPROTO_TCP;
        assert(bpf_map_lookup_elem(map_fd[0], &key, &tcp_cnt) == 0);
        ...
        sleep(1);
    }

    return 0;
}
```

BPF的前端BCC

BCC

- 是什么？ → BPF编译器合集（BPF Compiler Collection）
- 能干什么 → 它提供了一个编写内核BPF程序的C语言环境，同时还提供了其他高级语言（如python，Lua，和C++）环境来实现用户端接口
- BCC，bpftrace和IO Visor的关系？ → BCC和bpftrace都是BPF的两个前端，源代码不在内核代码仓库中，托管在github上的一个名为IO Visor的Linux基金会项目

BCC其实提供了一种使用BPF编程的框架，这套框架提供给我们一些用户接口，并且屏蔽掉了一些加载，编译的复杂环节，只需要运行写好的BCC脚本，BPF程序就可以工作。

使用BCC编程

BCC提供了一个编写内核BPF程序的C语言环境，同时还提供了其他高级语言(如python)环境来实现用户端接口

```
#!/usr/bin/env python
# coding=utf-8

from bcc import BPF

program = '''

int kprobe__sys_clone(void *ctx)
{
    /* 向kernel trace buffer(/sys/kernel/debug/tracing/trace_pipe)写入字符串 */
    bpf_trace_printk("hello, world!\\n");

    return 0;
}
'''


#load eBPF program
b = BPF(text = program) #实例化一个新的BPF对象b
b.trace_print()
```

BPF学习资料

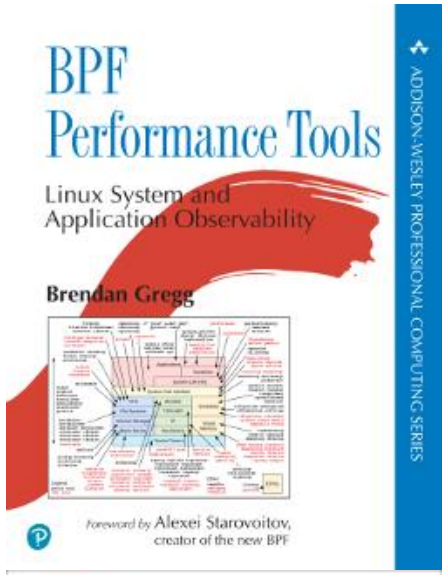
- 书籍
  - 《Linux内核观测技术BPF》
  - 《BPF之巅：洞悉Linux系统和应用性能》
  - 《Systems Performance》
  - 《BPF Performance Tools》

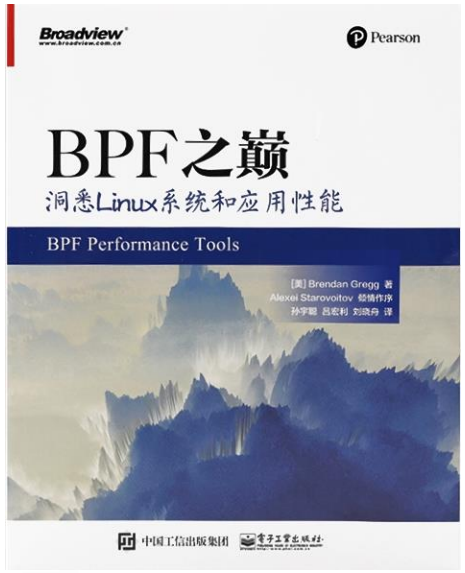
- Brendan Gregg大神的个人网站
  - http://www.brendangregg.com/index.html

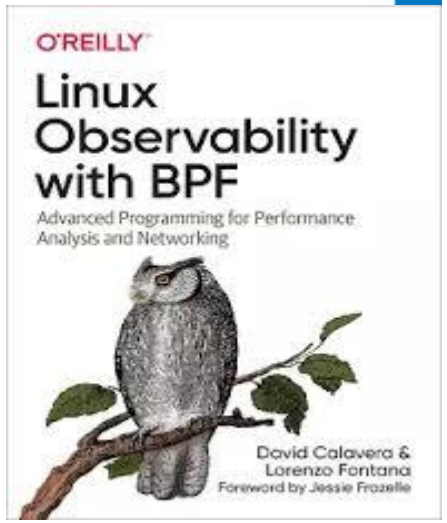
- Github
  - Linux基金会的IO Visor项目：https://github.com/iovisor
  - https://github.com/zoidbergwill/awesome-ebpf

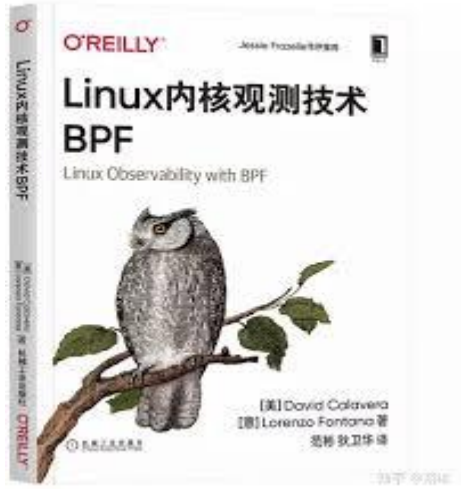
- 网站
  - Cilium eBPF：https://ebpf.io

- BPF原始论文
  - https://www.tcpdump.org/papers/bpf-usenix93.pdf









下面介绍Linux内核之旅的BPF观测可视化项目LMP

LMP

Linux microscope