Linux BPF

Фильтрация сокетов в Linux или пакетный фильтр Беркли (Berkeley Packet Filter или BPF). На основе перевода документа Documentation/networking/filter.rst из состава исходных кодов ядра Linux.

Введение

Фильтрация сокетов Linux (Linux Socket Filtering или LSF) основана на пакетных фильтрах Беркли. Несмотря на множество различий между фильтрацией в ядре BSD и Linux, при обсуждении BPF или LSF в контексте Linux рассматриваются практически те же механизмы фильтрации в ядре Linux.

BPF позволяет программам пользовательского пространства присоединять фильтры к любому сокету и разрешать или запрещать прохождение некоторых типов данных через сокет. LSF применяет такую же структуру кода фильтрации, что используется BPF в BSD, поэтому полезно прочесть руководство (man) BSD bpf.4 для работы с фильтрами.

В Linux фильтрация BPF существенно проще, чем в BSD - здесь не нужно беспокоиться об устройствах и других вещах. Нужно просто создать код своего фильтра, передать его ядру через опцию SO_ATTACH_FILTER и после успешной проверки кода ядром сразу же начнётся фильтрация данных на сокете. Фильтры сокетов можно отсоединять с помощью опции SO_DETACH_FILTER. Вероятно эта возможность не будет применяться часто, поскольку при закрытии сокета фильтры удаляются автоматически. Другим менее распространенным случаем является добавление другого фильтра на сокете, где фильтр уже работает. В этом случае ядро будет удалять прежний фильтр и устанавливать новый, если тот прошёл проверку. При отказе в процессе проверки на сокете сохранится прежний фильтр.

Опция SO_LOCK_FILTER позволяет заблокировать присоединённый к сокету фильтр и его нельзя будет удалить или изменить. Это позволяет процессу организовать сокет, связать с ним фильтр, заблокировать его, а затем сбросить привилегии, будучи уверенным в сохранении фильтра до момента закрытия сокета.

Основным пользователем этих конструкций является библиотека libpcap. Используя команды вида tcpdump -i em1 port 22, передаваемые через внутренний компилятор libpcap, который создаёт структуру, передаваемую в конечном итоге ядру через SO_ATTACH_FILTER, библиотека может устанавливать нужные фильтры.

Хотя здесь упоминались лишь сокеты, BPF в Linux применяется и во многих других местах. Это включает xt_bpf для netfilter, cls_bpf на уровне qdisc в ядре, SECCOMP-BPF (SECure COMPuting [1]) и пр.

Структура

Приложения пользовательского уровня включают файл <linux/filter.h>, содержащий указанную ниже структуру.

```
struct sock_filter {   /* Блок фильтра */
        __u16   code;  /* Фактический код фильтрации */
        __u8    jt;    /* Переход при совпадении */
        __u8    jf;    /* Переход при несовпадении */
        __u32   k;     /* Базовое многоцелевое поле */
};
```

Такая структура организуется в форме массива квартетов (4-tuple) в форме (code, jt, jf, k). Поля jt и jf задают смещение для перехода, а k указывает базовое значение для использования представленным ниже кодом.

```
struct sock_fprog {           /* Требуется для SO_ATTACH_FILTER. */
        unsigned short len;   /* Число блоков фильтра */
        struct sock_filter __user *filter;
};
```

Для фильтрации на сокете указатель на эту структуру (см. Пример) передаётся ядру через setsockopt().

Пример

```
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <linux/if_ether.h>
/* ... */

/* Для упомянутой выше команды tcpdump -i em1 port 22 -dd */
struct sock_filter code[] = {
        { 0x28,  0,  0, 0x0000000c },
        { 0x15,  0,  8, 0x000086dd },
        { 0x30,  0,  0, 0x00000014 },
        { 0x15,  2,  0, 0x00000084 },
        { 0x15,  1,  0, 0x00000006 },
        { 0x15,  0, 17, 0x00000011 },
        { 0x28,  0,  0, 0x00000036 },
        { 0x15, 14,  0, 0x00000016 },
        { 0x28,  0,  0, 0x00000038 },
        { 0x15, 12, 13, 0x00000016 },
        { 0x15,  0, 12, 0x00000800 },
        { 0x30,  0,  0, 0x00000017 },
        { 0x15,  2,  0, 0x00000084 },
        { 0x15,  1,  0, 0x00000006 },
        { 0x15,  0,  8, 0x00000011 },
        { 0x28,  0,  0, 0x00000014 },
        { 0x45,  6,  0, 0x00001fff },
        { 0xb1,  0,  0, 0x0000000e },
```

```
            { 0x48,  0,  0, 0x0000000e },
            { 0x15,  2,  0, 0x00000016 },
            { 0x48,  0,  0, 0x00000010 },
            { 0x15,  0,  1, 0x00000016 },
            { 0x06,  0,  0, 0x0000ffff },
            { 0x06,  0,  0, 0x00000000 },
    };

    struct sock_fprog bpf = {
            .len = ARRAY_SIZE(code),
            .filter = code,
    };

    sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    if (sock < 0)
            /* ... код ... */

    ret = setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf));
    if (ret < 0)
            /* ... код ... */

    /* ... */
    close(sock);
```

Приведённый выше код присоединяет фильтр к сокету PF_PACKET для отбора всех пакетов IPv4 и IPv6, направленных в порт 22. Остальные пакеты сокет будет отбрасывать.

Вызову setsockopt() для SO_DETACH_FILTER не требуется аргументов, а SO_LOCK_FILTER для блокировки фильтра принимает целое число 0 или 1.

Отметим, что фильтры могут применяться не только для сокетов PF_PACKET, но и для сокетов иных семейств.

Сводка системных вызовов приведена ниже

```
setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_FILTER, &val, sizeof(val));
setsockopt(sockfd, SOL_SOCKET, SO_DETACH_FILTER, &val, sizeof(val));
setsockopt(sockfd, SOL_SOCKET, SO_LOCK_FILTER,   &val, sizeof(val));
```

Обычно для большинства фильтров на пакетных сокетах будет применяться высокоуровневый синтаксис и разработчикам приложений следует придерживаться его. Остальное обеспечит библиотека libpcap.

В указанных ниже случаях могут применяться «написанные вручную» фильтры:

1. применение libpcap не рассматривается (невозможно);
2. требуемые фильтры BPF должны использовать расширения, не поддерживаемые компилятором libpcap;
3. фильтр слишком сложен и его поддержка компилятором libpcap не очевидна;
4. код конкретного фильтра следует оптимизировать не так, как это делает встроенный компилятор libpcap.

Например, у пользователей xt_bpf и cls_bpf могут возникать требования, приводящие к более сложному коду фильтрации, или не выражаемые с помощью libpcap (например, с разными кодами возврата для разных путей в коде). Кроме того, разработчики BPF JIT[1] могут захотеть вручную написать тесты, для чего потребуется доступ к коду BPF на низком уровне.

Машина и набор инструкций BPF

В bpf имеется инструмент bpf_asm, который можно применять для написания низкоуровневых фильтров, подобных описанным выше. Похожий на ассемблер синтаксис bpf_asm ниже используется в описаниях вместо явных кодов операций. Этот синтаксис очень похож на применяемый в статье Steven McCanne т Van Jacobson о фильтрах BPF [2].

Базовые элементы архитектуры BPF приведены в таблице.

Элемент	***Описание***
A	Аккумулятор (накопитель) размером 32 бита.
XX	Регистр X размером 32 бита.
M[]	Массив из 16 регистров (от 0 до 15) по 32 бита или хранилище в памяти (scratch memory store).

Программа, транслируемая bpf_asm в коды операций (opcode), представляет собой массив упомянутых элементов

```
  op:16, jt:8, jf:8, k:32
```

Элемент op является 16-битовым кодом операции, задающим конкретную инструкцию (команду). Элементы jt и jf являются 8-битовыми словами, указывающими адрес перехода для совпадения (true) и несовпадения (false), соответственно. Элемент k содержит аргумент, интерпретация которого зависит от операции op.

Набор инструкций (команд) включает команды загрузки (load), записи (store), ветвления (branch), арифметических операций (alu), копирования и возврата управления (return), которые представлены также в синтаксисе bpf_asm. В таблице указаны все доступные команды bpf_asm с указанием операций, заданных в файле linux/filter.h.

Инструкция	***Режим адресации***	***Описание***
Команды загрузки		
ld	1, 2, 3, 4, 12	Загрузка слова в A
ldi	4	Загрузка слова в A
ldh	1, 2	Загрузка полуслова в A
ldb	1, 2	Загрузка байта в A
ldx	3, 4, 5, 12	Загрузка слова в X
ldxi	4	Загрузка слова в X

[1] Just-in-Time - компиляция в нужный момент.

ldxb	5	Загрузка байта в X
Команды записи		
st	3	Запись (сохранение) A в M[]
stx	3	Запись (сохранение) X в M[]
Команды перехода		
jmp	6	Переход к метке
ja	6	Переход к метке
jeq	7, 8, 9, 10	Переход при A == <x>
jneq	9, 10	Переход при A != <x>
jne	9, 10	Переход при A != <x>
jlt	9, 10	Переход при A < <x>
jle	9, 10	Переход при A <= <x>
jgt	7, 8, 9, 10	Переход при A > <x>
jge	7, 8, 9, 10	Переход при A >= <x>
jset	7, 8, 9, 10	Переход при A & <x>
Арифметические операции		
add	0, 4	A + <x>
sub	0, 4	A - <x>
mul	0, 4	A * <x>
div	0, 4	A / <x>
mod	0, 4	A % <x>
neg		!A
and	0, 4	& <x>
or	0, 4	A | <x>
xor	0, 4	A ^ <x>
lsh	0, 4	A << <x>
rsh	0, 4	A >> <x>
Команды копирования		
tax		Копирование A в X
txa		Копирование X в A
Команда возврата		
ret	4, 11	Возврат управления

В следующей таблице указаны режимы адресации (второй столбец предыдущей таблицы).

Режим	***Синтаксис***	***Описание***
0	x/%x	Регистр X
1	[k]	Двоичное полуслово (BHW) с байтовым смещением k в пакете
2	[x + k]	Двоичное полуслово (BHW) со смещением X + k в пакете
3	M[k]	Слово со смещением k в массиве M[]
4	#k	Литеральное значение, хранящееся в k
5	4*([k]&0xf)	Младший полубайт (nibble) * 4 в байте со смещением k в пакете
6	L	Переход к метке L
7	#k,Lt,Lf	Переход к метке Lt при совпадении (true), иначе переход к Lf
8	x/%x,Lt,Lf	Переход к метке Lt при совпадении (true), иначе переход к Lf
9	#k,Lt	Переход к Lt, если утверждение (predicate) верно (true)
10	x/%x,Lt	Переход к Lt, если утверждение (predicate) верно (true)
11	a/%a	Аккумулятор A
12	расширение	Расширение BPF

Ядро Linux включает расширения BPF, применяемые с командами загрузки, путём указания аргумента k с отрицательным смещением и смещением конкретного расширения. Результат таких расширений BPF загружается в A. Возможные расширения BPF показаны в таблице.

Расширение	***Описание***
len	skb->len
proto	skb->protocol
type	skb->pkt_type
poff	Смещение начала данных (payload)
ifidx	skb->dev->ifindex
nla	Атрибут netlink типа X со смещением A
nlan	Вложенный атрибут netlink типа X со смещением A
mark	skb->mark
queue	skb->queue_mapping
hatype	skb->dev->type
rxhash	skb->hash
cpu	raw_smp_processor_id()
vlan_tci	skb_vlan_tag_get(skb)
vlan_avail	skb_vlan_tag_present(skb)
vlan_tpid	skb->vlan_proto
rand	prandom_u32()

Эти расширения могут также включать префикс #.

Примеры низкоуровневых BPF

Пакеты ARP

```
ldh [12]
jne #0x806, drop
ret #-1
drop: ret #0
```

Пакеты IPv4 TCP

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #6, drop
ret #-1
drop: ret #0
```

Выборка случайного пакета ICMP, 1 из 4

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #1, drop
# Получение случайного значения uint32
ld rand
mod #4
jneq #1, drop
ret #-1
drop: ret #0
```

Пример фильтра SECCOMP

```
ld [4]                  /* offsetof(struct seccomp_data, arch) */
jne #0xc000003e, bad    /* AUDIT_ARCH_X86_64 */
ld [0]                  /* offsetof(struct seccomp_data, nr) */
jeq #15, good           /* __NR_rt_sigreturn */
jeq #231, good          /* __NR_exit_group */
jeq #60, good           /* __NR_exit */
jeq #0, good            /* __NR_read */
jeq #1, good            /* __NR_write */
jeq #5, good            /* __NR_fstat */
jeq #9, good            /* __NR_mmap */
jeq #14, good           /* __NR_rt_sigprocmask */
jeq #13, good           /* __NR_rt_sigaction */
jeq #35, good           /* __NR_nanosleep */
bad: ret #0             /* SECCOMP_RET_KILL_THREAD */
good: ret #0x7fff0000   /* SECCOMP_RET_ALLOW */
```

Примеры низкоуровневых расширений BPF

Пакет для интерфейса с индексом 13

```
ld ifidx
jneq #13, drop
ret #-1
drop: ret #0
```

VLAN с идентификатором 10

```
ld vlan_tci
jneq #10, drop
ret #-1
drop: ret #0
```

Приведённые выше примеры кода можно поместить в файл (здесь назван foo) и затем передать bpf_asm для генерации кодов операций, которые будут понятны xt_bpf и cls_bpf и могут загружаться в них напрямую. Например, для указанного выше фильтра ARP результат будет иметь вид

```
$ ./bpf_asm foo
4,40 0 0 12,21 0 1 2054,6 0 0 4294967295,6 0 0 0,
```

Или при выводе в стиле копирования и вставки C

```
$ ./bpf_asm -c foo
{ 0x28,  0,  0, 0x0000000c },
{ 0x15,  0,  1, 0x00000806 },
{ 0x06,  0,  0, 0xffffffff },
{ 0x06,  0,  0, 0000000000 },
```

В частности, поскольку применение с xt_bpf или cls_bpf может приводить к более сложным фильтрам BPF, которые поначалу могут казаться неочевидными, рекомендуется тестировать фильтры перед их установкой в работающей системе. Для этого служит простой инструмент bpf_dbg, исходный код которого помещён в каталог tools/bpf/ исходных кодов ядра. Этот отладчик позволяет протестировать фильтры BPF на указанных файлах pcap, организовать пошаговое выполнение кода BPF на файлах pcap и получить содержимое регистров машины BPF.

Для запуска отладчика bpf_dbg служит команда

```
# ./bpf_dbg
```

Если ввод и вывод отличается от стандартного (stdin, stdout), bpf_dbg принимает замену stdin в качестве первого аргумента и stdout - в качестве второго. Например, ./bpf_dbg test_in.txt test_out.txt. Кроме того, можно задать конфигурацию libreadline в файле ~/.bpf_dbg_init, а история команд сохраняется в файле ~/.bpf_dbg_history.

Взаимодействие в bpf_dbg происходит через командный процессор (shell), поддерживающий дополнение команд. В приведённых ниже примерах команды вводятся из оболочки bpf_dbg. Примеры работы приведены ниже.

load bpf 6,40 0 0 12,21 0 3 2048,48 0 0 23,21 0 1 1,6 0 0 65535,6 0 0 0

Фильтр BPF загружается из стандартного вывода bpf_asm или преобразуется из команды, например, ``tcpdump -iem1 -ddd port 22 | tr '\n' ','`. Отметим, что для отладки JIT (Компилятор JIT) эта команда создаёт временный сокет и загружает код BPF в ядро, поэтому она будет полезна и для разработчиков JIT.

load pcap foo.pcap

Загрузка стандартного файла tcpdump pcap.

run [<n>]

bpf passes:1 fails:9

Работает для всех пакетов из файла pcap, указывая число соответствующих и не соответствующих фильтру пакетов. Можно задать число обрабатываемых пакетов из файла.

disassemble::

```
l0:     ldh [12]
l1:     jeq #0x800, l2, l5
l2:     ldb [23]
l3:     jeq #0x1, l4, l5
l4:     ret #0xffff
l5:     ret #0
```

Выводит дизассемблированный код BPF.

dump::

```
/* { op, jt, jf, k }, */
{ 0x28,  0,  0, 0x0000000c },
{ 0x15,  0,  3, 0x00000800 },
{ 0x30,  0,  0, 0x00000017 },
{ 0x15,  0,  1, 0x00000001 },
{ 0x06,  0,  0, 0x0000ffff },
{ 0x06,  0,  0, 0000000000 },
```

Выводит дамп кода BPF в стиле C.

breakpoint 0::

breakpoint at: l0: ldh [12]

breakpoint 1::

breakpoint at: l1: jeq #0x800, l2, l5

Команду задают точку прерывания на конкретных инструкциях BPF. При вводе команды run исполнение будет проходить через файл pcap от текущего пакета до заданной точки прерывания. Последующая команда run будет выполняться от текущей активной точки прерывания.

run::

```
-- register dump --
pc:       [0]                       <-- счётчик команд
code:     [40] jt[0] jf[0] k[12]    <-- код BPF для текущей инструкции
curr:     l0:  ldh [12]             <-- дизассемблированная текущая инструкция
A:        [00000000][0]             <-- содержимое A (hex, decimal)
X:        [00000000][0]             <-- содержимое X (hex, decimal)
M[0,15]:  [00000000][0]             <-- содержимое M (hex, decimal)
-- packet dump --                   <-- текущий пакет от pcap (hex)
len: 42
    0: 00 19 cb 55 55 a4 00 14 a4 43 78 69 08 06 00 01
16: 08 00 06 04 00 01 00 14 a4 43 78 69 0a 3b 01 26
32: 00 00 00 00 00 00 0a 3b 01 01
(breakpoint)
>
```

breakpoint::

```
breakpoints: 0 1
```

Выводит список установленных точек прерывания.

step [-<n>, +<n>]

Пошаговое выполнение программы BPF от текущего смещения. При каждом вызове step выводится показанный выше дамп регистров. Исполнение может происходить также в прямом и обратном направлении.

select <n>

Выбирает заданный пакет из файла pcap для работы с ним. При следующем вызова run или step программа BPF будет обрабатывать заданный пользователем пакет. Нумерация пакетов начинается с 1, как в Wireshark.

quit

Завершает работу bpf_dbg.

Компилятор JIT

Ядро Linux включает компилятор BPF JIT для платформ x86_64, SPARC, PowerPC, ARM, ARM64, MIPS, RISC-V, s390, который включается параметром конфигурации CONFIG_BPF_JIT. Компилятор JIT автоматически вызывается для каждого подключённого фильтра из пользовательского пространства или для внутренних пользователей, если это было разрешено командой

```
echo 1 > /proc/sys/net/core/bpf_jit_enable
```

Для разработчиков JIT, выполняющих аудит и другие операции, при каждом запуске компилятора можно задать вывод созданного кода в журнал ядра с помощью команды

```
echo 2 > /proc/sys/net/core/bpf_jit_enable
```

Ниже приведён пример вывода с помощью dmesg.

```
[ 3389.935842] flen=6 proglen=70 pass=3 image=ffffffffa0069c8f
[ 3389.935847] JIT code: 00000000: 55 48 89 e5 48 83 ec 60 48 89 5d f8 44 8b 4f 68
[ 3389.935849] JIT code: 00000010: 44 2b 4f 6c 4c 8b 87 d8 00 00 00 be 0c 00 00 00
[ 3389.935850] JIT code: 00000020: e8 1d 94 ff e0 3d 00 08 00 00 75 16 be 17 00 00
[ 3389.935851] JIT code: 00000030: 00 e8 28 94 ff e0 83 f8 01 75 07 b8 ff ff 00 00
[ 3389.935852] JIT code: 00000040: eb 02 31 c0 c9 c3
```

При включённой опции CONFIG_BPF_JIT_ALWAYS_ON для bpf_jit_enable устанавливается значение 1 и попытка поменять это значение приведёт к отказу. Это происходит и при установке bpf_jit_enable = 2, поскольку не рекомендуется выводить финальный дамп JIT в журнал ядра. Взамен рекомендуется выполнять самоанализ с помощью bpftool (из каталога tools/bpf/bpftool/).

В каталоге исходных кодов ядра tools/bpf/ имеется инструмент bpf_jit_disasm[1] для вывода дизассемблированных дампов из журнала ядра.

[1] В новых версиях ядра он называется jit_disasm.

```
# ./bpf_jit_disasm
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
0:      push   %rbp
1:      mov    %rsp,%rbp
4:      sub    $0x60,%rsp
8:      mov    %rbx,-0x8(%rbp)
c:      mov    0x68(%rdi),%r9d
10:     sub    0x6c(%rdi),%r9d
14:     mov    0xd8(%rdi),%r8
1b:     mov    $0xc,%esi
20:     callq  0xffffffffe0ff9442
25:     cmp    $0x800,%eax
2a:     jne    0x0000000000000042
2c:     mov    $0x17,%esi
31:     callq  0xffffffffe0ff945e
36:     cmp    $0x1,%eax
39:     jne    0x0000000000000042
3b:     mov    $0xffff,%eax
40:     jmp    0x0000000000000044
42:     xor    %eax,%eax
44:     leaveq
45:     retq
```

Опция -o задаёт «аннотирование» кодов ассемблерных команд, которое может быть полезно для разработчиков JIT.

```
# ./bpf_jit_disasm -o
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
0:      push   %rbp
        55
1:      mov    %rsp,%rbp
        48 89 e5
4:      sub    $0x60,%rsp
        48 83 ec 60
8:      mov    %rbx,-0x8(%rbp)
        48 89 5d f8
c:      mov    0x68(%rdi),%r9d
        44 8b 4f 68
10:     sub    0x6c(%rdi),%r9d
        44 2b 4f 6c
14:     mov    0xd8(%rdi),%r8
        4c 8b 87 d8 00 00 00
1b:     mov    $0xc,%esi
        be 0c 00 00 00
20:     callq  0xffffffffe0ff9442
        e8 1d 94 ff e0
25:     cmp    $0x800,%eax
        3d 00 08 00 00
2a:     jne    0x0000000000000042
        75 16
2c:     mov    $0x17,%esi
        be 17 00 00 00
31:     callq  0xffffffffe0ff945e
        e8 28 94 ff e0
36:     cmp    $0x1,%eax
        83 f8 01
39:     jne    0x0000000000000042
        75 07
3b:     mov    $0xffff,%eax
        b8 ff ff 00 00
40:     jmp    0x0000000000000044
        eb 02
42:     xor    %eax,%eax
        31 c0
44:     leaveq
        c9
45:     retq
        c3
```

Для разработчиков BPF JIT инструменты bpf_jit_disasm, bpf_asm и bpf_dbg обеспечивают средства разработки и тестирования компиляторов JIT в ядре.

Компоненты BPF в ядре

Внутри интерпретатора в ядре применяется иной формат набора инструкций с принципами, аналогичными базовым принципам BPF, описанным выше. Однако формат набора инструкций смоделирован ближе к базовой архитектуре, чтобы имитировать естественный набор инструкций для повышения производительности (см. ниже). Эту новую архитектуру ISA[2] называют eBPF или internal BPF. eBPF является сокращением от [e]xtended BPF, что отличается от расширений BPF. eBPF представляет собой ISA, тогда как расширения BPF обозначают «классические» фильтры BPF с дополнением инструкций BPF_LD | BPF_{B,H,W} | BPF_ABS.

Набор инструкций рассчитан на компиляцию JIT с взаимно-однозначным отображением, что также может позволить использовать компиляторы GCC/LLVM для генерации оптимизированного кода eBPF с помощью eBPF backend, работающих почти так же быстро, как естественный код.

[2]instruction-set architecture.

Новый набор инструкций исходно создавался для написания программ в стиле языка C с ограничениями и компиляции в eBPF с помощью дополнительного бэкэнда GCC/LLVM, чтобы можно было в нужный момент (just-in-time) выполнить отображение на современные 64-битовые CPU с минимальными потерями производительности в два этапа (C -> eBPF -> естественный код).

В настоящее время новый формат применяется для запуска пользовательских программ BPF, включая seccomp BPF, классические фильтры сокетов, классификаторы трафика cls_bpf, классификаторы групповых драйверов (team driver) для режима распределения нагрузки, расширений netfilter xt_bpf, диссекторов и классификаторов PTP и т. п. Все они преобразуются ядром в представление нового набора инструкций и выполняются интерпретатором eBPF. Для обработчиков в ядре это выполняется автоматически с помощью вызовов bpf_prog_create() для установки фильтров и bpf_prog_destroy() для их удаления. Функция bpf_prog_run(filter, ctx) автоматически вызывает интерпретатор eBPF или скомпилированный JIT код для работы фильтра. Параметр filter задаёт указатель на структуру bpf_prog, полученный от bpf_prog_create(), а ctx - текущий контекст (например, указатель на skb). Все ограничения для bpf_check_classic() применяются до преобразования к новой схеме.

В настоящее время классический формат BPF применяется для компиляции JIT на большинстве 32-битовых архитектур, а для x86-64, aarch64, s390x, powerpc64, sparc64, arm32, riscv64, riscv32 выполняется компиляция JIT из набора инструкций eBPF.

Основные изменения внутреннего формата перечислены ниже.

- *Число регистров увеличено с 2 до 10.*

  В старом формате применялось 2 регистра (A и X), а также скрытый указатель на кадр. Новая схема имеет 10 внутренних регистров и доступный лишь для чтения указатель на кадр. Поскольку в 64-битовых CPU аргументы передаются функциям через регистры, число аргументов, передаваемых из программы eBPF во внутреннюю функцию ядра, ограничено пятью (5) и 1 регистр служит для возвращаемого функцией значения. Процессоры x86_64 передают первые 6 аргументов в регистрах, а aarch64, sparcv9, mips64 имеют 7 или 8 регистров для аргументов. В x86_64 имеется 6 сохраняемых вызываемой функцией регистров, а aarch64, sparcv9, mips64 - не менее 11.

  Соглашения для вызовов eBPF указаны ниже.

  - R0 - значение, возвращаемое внутренней функцией ядра, и код завершения для программы eBPF.
  - R1 - R5 - аргументы, передаваемые из программы eBPF внутренней функции ядра.
  - R6 - R9 - сохраняемые вызываемой функцией регистры.
  - R10 - доступный лишь для чтения указатель на стек доступа.

  Таким образом, для всех регистров eBPF обеспечивается взаимно-однозначное отображение на аппаратные (HW) регистры x86_64, aarch64 и т. п., а соглашения о вызовах eBPF напрямую отображаются на ABI[1], используемые ядром 64-битовых платформ.

  На 32-битовых платформах JIT может отображать программы, использующие лишь 32-разрядную арифметику, и может поддерживать интерпретацию более сложных программ.

  R0 - R5 - это временные (scratch) регистры и программ eBPF должна при необходимости заполнять их при вызовах. Отметим, что существует лишь одна программа eBPF (eBPF main) и она не может вызывать другие функции eBPF, однако может обращаться ко встроенным функциям ядра.

- *Регистры расширены с 32 битов до 64.*

  Семантика исходных 32-битовых операций ALU сохранена за счёт применения 32-битовых субрегистров. Все регистры eBPF являются 64-битовыми и младшие 32 бита служат субрегистром, а старшие заполняются нулями при записи в них. Это напрямую работает в архитектуре x86_64 и arm64, но для других JIT более сложно.

  На 32-битовых платформах 64-битовые внутренние программы BPF выполняются через интерпретатор. Их компиляторы JIT могут конвертировать программы BPF, использующие лишь 32-битовые субрегистры к естественному набору инструкций, а остальная часть интерпретируется.

  Операции являются 64-битовыми, поскольку на 64-битовых платформах указатели также имеют размер 64 бита и нужно передавать 64-битовые значения в функции ядра и из них, в ином случае 32-битовые регистры eBPF потребовали бы определений ABI для регистровых пар, что не позволило бы напрямую сопоставлять регистры eBPF с аппаратными регистрами и компилятору JIT потребовались бы операции объединения, расщепления и перемещения для каждого регистра при передаче в функцию или из неё, что было бы сложно, подвержено ошибкам и медленно. Другой причиной является применение 64-битовых атомарных счётчиков.

- *Условные цели jt/jf заменены на jt/fall-through.*

  Конструкции вида «if (cond) jump_true; else jump_false;» в исходном решении заменены на конструкции вида «if (cond) jump_true; /* else fall-through */».

- *Введены bpf_call insn и соглашение о передаче регистров для вызовов с нулевыми издержками при передаче в другие функции ядра и из них.*

  Перед вызовом встроенной функции ядра внутренней программе BPF нужно поместить аргументы функции в регистры R1 - R5 для выполнения соглашения о вызовах, после чего интерпретатор возьмёт значения из регистров и передаст их внутренней функции ядра. Если регистры R1 - R5 отображены на регистры CPU, используемые архитектурой для передачи аргументов, компилятору JIT не нужно выполнять дополнительные операции. Аргументы функции будут в нужных регистрах и инструкция BPF_CALL будет компилироваться JIT в

[1]Application binary interface - двоичный интерфейс с приложениями.

одну аппаратную инструкцию вызова (call). Такое соглашение о вызовах было выбрано для охвата распространённых случаев без снижения производительности.

После вызова встроенной функции ядра регистры R1 - R5 сбрасываются в нечитаемое состояние, а R0 содержит код возврата функции. Поскольку регистры R6 - R9 сохраняет вызываемая функция, они не меняются в результате вызова.

В качестве примера рассмотрим 3 приведённых ниже функций C.

```
u64 f1() { return (*_f2)(1); }
u64 f2(u64 a) { return f3(a + 1, a); }
u64 f3(u64 a, u64 b) { return a - b; }
```

Компилятор GCC может преобразовать f1, f3 в команды x86_64

```
f1:
    movl $1, %edi
    movq _f2(%rip), %rax
    jmp  *%rax
f3:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

Функция f2 в eBPF может иметь вид

```
f2:
    bpf_mov R2, R1
    bpf_add R1, 1
    bpf_call f3
    bpf_exit
```

Если f2 компилируется JIT и указатель сохраняется в _f2, вызовы и возврат f1 -> f2 -> f3 произойдут «без стыков» (seamless). Без JIT требуется интерпретатор __bpf_prog_run() для вызова f2.

По практическим причинам все программы eBPF имеют единственный аргумент ctx, который уже помещён в R1 (например, при запуске __bpf_prog_run()) и программа может вызывать функции ядра с числом аргументов до 5. Использование 6 и более аргументов в настоящее время не поддерживается, но это ограничение может быть снято в будущем.

На 64-битовых платформах все регистры взаимно-однозначно сопоставляются с аппаратными регистрами. Например, компилятор x86_64 JIT может отображать их как показано ниже.

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

Архитектура x86_64 ABI требует использовать rdi, rsi, rdx, rcx, r8, r9 для передачи аргументов, а rbx, r12 - r15 - для сохранения вызываемой функцией.

Приведённый ниже код внутренней псевдопрограммы BPF

```
bpf_mov R6, R1 /* сохранение ctx */
bpf_mov R2, 2
bpf_mov R3, 3
bpf_mov R4, 4
bpf_mov R5, 5
bpf_call foo
bpf_mov R7, R0 /* сохранение значения, возвращаемого foo() */
bpf_mov R1, R6 /* восстановление ctx для следующего вызова */
bpf_mov R2, 6
bpf_mov R3, 7
bpf_mov R4, 8
bpf_mov R5, 9
bpf_call bar
bpf_add R0, R7
bpf_exit
```

после JIT для x86_64 может иметь вид

```
push %rbp
mov %rsp,%rbp
sub $0x228,%rsp
mov %rbx,-0x228(%rbp)
mov %r13,-0x220(%rbp)
mov %rdi,%rbx
mov $0x2,%esi
mov $0x3,%edx
mov $0x4,%ecx
mov $0x5,%r8d
callq foo
mov %rax,%r13
mov %rbx,%rdi
mov $0x6,%esi
```

```
    mov $0x7,%edx
    mov $0x8,%ecx
    mov $0x9,%r8d
    callq bar
    add %r13,%rax
    mov -0x228(%rbp),%rbx
    mov -0x220(%rbp),%r13
    leaveq
    retq
```

Код этого примера эквивалентен фрагменту C, показанному ниже.

```
    u64 bpf_filter(u64 ctx)
    {
        return foo(ctx, 2, 3, 4, 5) + bar(ctx, 6, 7, 8, 9);
    }
```

Встроенные функции ядра foo() и bar() с прототипом u64 (*)(u64 arg1, u64 arg2, u64 arg3, u64 arg4, u64 arg5); будут получать аргументы в соответствующих регистрах и помещают возвращаемые значения в %rax (регистр R0 в eBPF). Пролог и эпилог функций создаются JIT и являются неявными в интерпретаторе. Регистры R0-R5 являются временными (scratch). Поэтому программа eBPF должна сохранять их при вызовах в соответствии с соглашением о вызовах. Например, приведённая ниже программа является некорректной.

```
    bpf_mov R1, 1
    bpf_call foo
    bpf_mov R0, R1
    bpf_exit
```

После вызова регистры R1-R5 содержат «хлам» и не могут быть прочитаны. Для проверки внутренних программ BPF служит встроенный в ядро блок проверки eBPF (verifier).

В новом решении eBPF поддерживает не более 4096 insn, это означает, что любая программа будет завершаться быстро и вызовет лишь фиксированное число функций ядра. Исходный формат BPF и новый формат используют команды с 2 операндами, что помогает выполнить взаимно-однозначное сопоставление между eBPF insn и x86 insn при компиляции JIT.

Указатель входного контекста для вызова функции интерпретатора является базовым и его содержимое определяется конкретным вариантом применения. Для seccomp регистр R1 указывает на seccomp_data, для преобразованных фильтров BPF - на skb.

Программа с внутренней трансляцией состоит из указанных ниже элементов

```
  op:16, jt:8, jf:8, k:32    ==>    op:8, dst_reg:4, src_reg:4, off:16, imm:32
```

На данный момент реализовано 87 внутренних инструкций BPF. 8-битовый код операции op позволяет определять новые команды. Некоторые инструкции могут использовать 16-, 24- и 32-битовое кодирование. Для совместимости с имеющимися инструкциями новые инструкции должны иметь размер, кратный 8 битам.

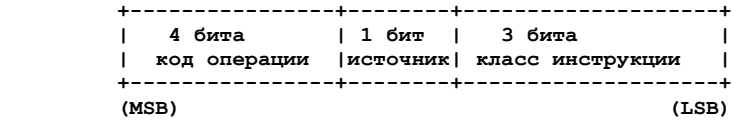
Внутренний BPF представляет собой набор инструкций общего назначения RISC. Не все регистры и инструкции применяются при трансляции из исходного BPF в новый формат. Например, фильтры сокетов не используют инструкцию exclusive add, а фильтры трассировки могут поддерживать счётчики событий. Регистр R9 не используется фильтрами сокетов, но более сложным фильтрам может не хватать регистров и они должны будут использовать стек.

Внутренний BPF можно применять в качестве ассемблера общего назначения для финального этапа оптимизации производительности и фильтры сокетов и seccomp используют его как ассемблер. Фильтры трассировки могут применять его в качестве ассемблера для генерации кода из ядра. Использование в ядре может быть не привязано к соображениям безопасности, поскольку генерируемый код внутреннего BPF может оптимизировать путь внутреннего кода и не раскрываться в пользовательское пространство. Безопасность внутреннего BPF может контролироваться блоком проверки (verifier). В случаях, подобных описанным его можно применять как защищённый набор инструкций.

Как и исходный BPF, новый формат работает в контролируемой среде, является детерминированным и ядро может легко доказать это. Безопасность программы можно определить за 2 шага (Проверка eBPF). Сначала выполняется поиск в глубину для исключения циклов и других проверок CFG, а второй шаг начинается с первого insn и проходит по всем возможным путям. Это имитирует выполнение всех insn с наблюдением смены состояний регистров и стека.

Представление операций eBPF

В eBPF используется большинство классических кодов операций для упрощения преобразования BPF в eBPF. Для переходов и арифметических операций 8-битовое поле code поделено на три части, как показано на рисунке.



Три младших (LSB) бита задают класс инструкции, как показано в таблице.

Классы традиционного BPF		*Классы eBPF*	
BPF_LD	0x00	BPF_LD	0x00
BPF_LDX	0x01	BPF_LDX	0x01
BPF_ST	0x02	BPF_ST	0x02
BPF_STX	0x03	BPF_STX	0x03
BPF_ALU	0x04	BPF_ALU	0x04
BPF_JMP	0x05	BPF_JMP	0x05
BPF_RET	0x06	BPF_JMP32	0x06
BPF_MISC	0x07	BPF_ALU64	0x07

Для классов BPF_ALU и BPF_JMP четвёртый бит представляет источник - BPF_K = 0x00 или BPF_X = 0x08.

- В классическом BPF это означает:

```
BPF_SRC(code) == BPF_X - операндом-источником служит регистр X;
BPF_SRC(code) == BPF_K - операндом-источником является следующее 32-битовое значение.
```

- В eBPF это означает:

```
BPF_SRC(code) == BPF_X - операндом-источником служит регистр src_reg;
BPF_SRC(code) == BPF_K - операндом-источником является следующее 32-битовое значение.
```

4 старших (MSB) бита задают код операции.

Для классов BPF_ALU и BPF_ALU64 (в eBPF) коды BPF_OP(code) приведены ниже.

```
BPF_ADD   0x00
BPF_SUB   0x10
BPF_MUL   0x20
BPF_DIV   0x30
BPF_OR    0x40
BPF_AND   0x50
BPF_LSH   0x60
BPF_RSH   0x70
BPF_NEG   0x80
BPF_MOD   0x90
BPF_XOR   0xa0
BPF_MOV   0xb0  /* только eBPF - перенос из регистра в регистр */
BPF_ARSH  0xc0  /* только eBPF - добавление знака со сдвигом вправо */
BPF_END   0xd0  /* только eBPF - смена порядка битов */
```

Для классов BPF_JMP и BPF_JMP32 (в eBPF) коды BPF_OP(code) приведены ниже.

```
BPF_JA    0x00  /* только BPF_JMP */
BPF_JEQ   0x10
BPF_JGT   0x20
BPF_JGE   0x30
BPF_JSET  0x40
BPF_JNE   0x50  /* только eBPF - jump != */
BPF_JSGT  0x60  /* только eBPF - signed '>' */
BPF_JSGE  0x70  /* только eBPF - signed '>=' */
BPF_CALL  0x80  /* только eBPF BPF_JMP - вызов функции */
BPF_EXIT  0x90  /* только eBPF BPF_JMP - возврат из функции */
BPF_JLT   0xa0  /* только eBPF - unsigned '<' */
BPF_JLE   0xb0  /* только eBPF - unsigned '<=' */
BPF_JSLT  0xc0  /* только eBPF - signed '<' */
BPF_JSLE  0xd0  /* только eBPF - signed '<=' */
```

Таким образом, BPF_ADD | BPF_X | BPF_ALU означает 32-битовое сложение в BPF и eBPF. В BPF имеется лишь 2 регистра и это означает A += X, а в eBPF это означает dst_reg = (u32) dst_reg + (u32) src_reg. Точно так же, BPF_XOR | BPF_K | BPF_ALU означает A ^= imm32[1] в BPF и src_reg = (u32) src_reg ^ (u32) imm32 в eBPF.

В BPF применяется класс BPF_MISC для представления переносов A = X и X = A, а в eBPF для этого служит код BPF_MOV | BPF_X | BPF_ALU. Поскольку в eBPF нет операций BPF_MISC, класс 7 применяется для операций BPF_ALU64, которые аналогичны BPF_ALU, но выполняются с 64-битовыми операндами. Таким образом, BPF_ADD | BPF_X | BPF_ALU64 указывает 64-битовое сложение, т. е. dst_reg = dst_reg + src_reg

В BPF класс BPF_RET представлен одной операцией ret. Классический код BPF_RET | BPF_K означает копирование значения imm32 в регистр возврата и завершение функции. eBPF моделируется в соответствии с CPU, поэтому код BPF_JMP | BPF_EXIT в eBPF означает лишь выход из функции. Программа eBPF должна сохранить возвращаемое значение в регистре R0 перед выполнением BPF_EXIT. Класс 6 в eBPF применяется как BPF_JMP32 для обозначения тех же операций, что и BPF_JMP, но со сравнением 32-битовых операндов.

Для команд загрузки и сохранения 8-битовое поле code поделено на три части, как показано на рисунке.

```
+--------+--------+-------------------+
| 3 бита | 2 бита |   3 бита          |
| режим  | размер | класс инструкции  |
+--------+--------+-------------------+
(MSB)                             (LSB)
```

Возможные размеры указаны ниже.

```
BPF_W   0x00    /* слово */
BPF_H   0x08    /* полуслово */
BPF_B   0x10    /* байт */
BPF_DW  0x18    /* только eBPF - двойное слово */
```

Это поле определяет размер загружаемого или сохраняемого значения

```
B  - 1 байт
H  - 2 байта
W  - 4 байта
DW - 8 байт (только eBPF)
```

Поле режима может принимать одно из указанных ниже значений.

```
BPF_IMM     0x00  /* перенос 32 битов в BPF и 64 в eBPF */
BPF_ABS     0x20
BPF_IND     0x40
BPF_MEM     0x60
BPF_LEN     0x80  /* только BPF, резерв в eBPF */
BPF_MSH     0xa0  /* только BPF, резерв в eBPF */
BPF_ATOMIC  0xc0  /* только eBPF - неделимые операции */
```

В eBPF инструкции (BPF_ABS | <size> | BPF_LD) и (BPF_IND | <size> | BPF_LD) служат для доступа к данным пакета. Это пришлось перенести из BPF для обеспечения высокой производительности фильтров сокетов при работе в

[1] Следующие 32 бита.

интерпретаторе eBPF. Эти команды можно применять лишь в тех случаях, когда контекст интерпретатора является указателем на struct sk_buff и имеет 7 неявных операндов. Регистр R6 является неявным вводом, который должен содержать указатель на sk_buff, R0 является неявным выводом, содержащим извлечённые из пакета данные. Регистры R1 - R5 являются вспомогательными и недопустимо использовать их для сохранения данных при исполнении инструкции BPF_ABS | BPF_LD или BPF_IND | BPF_LD.

Эти инструкции имеют также неявное условие выхода из программы. При попытке программы eBPF обратиться к данным за пределами пакета интерпретатор будет прерывать исполнение программы. Поэтому компиляторы JIT должны сохранять это свойство. Поля src_reg и imm32 содержат явные входные данные для этих команд. Например, BPF_IND | BPF_W | BPF_LD означает

```
R0 = ntohl(*(u32 *) (((struct sk_buff *) R6)->data + src_reg + imm32))
```

Содержимое регистров R1 - R5 не сохраняется.

В отличие от набора команд BPF в eBPF включены базовые операции загрузки и сохранения (load/store)

```
BPF_MEM | <size> | BPF_STX:  *(size *) (dst_reg + off) = src_reg
BPF_MEM | <size> | BPF_ST:   *(size *) (dst_reg + off) = imm32
BPF_MEM | <size> | BPF_LDX:  dst_reg = *(size *) (src_reg + off)
```

Поле size в этих командах может принимать значение BPF_B, BPF_H, BPF_W или BPF_DW.

Включены также неделимые (atomic) операции, использующие поле imm для дополнительного кодирования

```
.imm = BPF_ADD, .code = BPF_ATOMIC | BPF_W  | BPF_STX: lock xadd *(u32 *)(dst_reg + off16) += src_reg
.imm = BPF_ADD, .code = BPF_ATOMIC | BPF_DW | BPF_STX: lock xadd *(u64 *)(dst_reg + off16) += src_reg
```

Поддерживаемые базовые неделимые операции указаны ниже.

```
BPF_ADD
BPF_AND
BPF_OR
BPF_XOR
```

Каждая из этих команд имеет семантику, эквивалентную BPF_ADD, т. е. адрес в памяти, указанный dst_reg + off, неделимо (atomically) изменяется с использованием src_reg в качестве другого операнда. Если флаг BPF_FETCH установлен напрямую, эти операции также переписывают src_reg значением, которое было в памяти до её изменения.

Более специализированная операция BPF_XCHG неделимо меняет src_reg значением, указанным адресом dst_reg + off. Операция BPF_CMPXCHG выполняет неделимое сравнение значения, указанного адресом dst_reg + off, со значением R0 и при совпадении заменяет его значением src_reg. Во всех случаях прежнее значение дополняется нулями и загружается обратно в R0.

Отметим, что 1- и 2-байтовые неделимые (atomic) операции не поддерживаются.

Clang может генерировать неделимые инструкции по умолчанию, если задана опция -mcpu=v3. При установке меньшей версии для -mcpu Clang может генерировать единственную неделимую инструкцию BPF_ADD без BPF_FETCH. Если нужно нужно включить неделимые операции при малых значениях версии -mcpu, можно использовать опции -Xclang -target-feature -Xclang +alu32.

BPF_XADD является устаревшим именем BPF_ATOMIC, указывающим операцию exclusive-add (монопольное сложение) при сброшенном (0) поле immediate.

В eBPF имеется инструкция BPF_LD | BPF_DW | BPF_IMM, состоящая из двух последовательных 8-байтовых блоков struct bpf_insn, интерпретируемых как загрузка непосредственного 64-битового значение в dst_reg. В BPF имеется похожая инструкция BPF_LD | BPF_W | BPF_IMM, загружающая непосредственное 32-битовое значение в регистр.

Проверка eBPF

Безопасность программы eBPF проверяется в два этапа.

Сначала выполняется проверка ациклического направленного графа (DAG[1]) для исключения петель и другие проверки CFG. В частности обнаруживаются программы с недоступными инструкциями (хотя проверка традиционного BPF разрешает их).

Второй этап начинается с первого insn и проходит по всем возможным путям, имитируя выполнение каждого insn и наблюдая смену состояний регистров и стека.

При старте программы регистр R1 содержит указатель на контекст и имеет тип PTR_TO_CTX. Если блок проверки insn, который делает R2=R1, регистр R2 также имеет тип тип PTR_TO_CTX и может использоваться в правой части выражений. Если R1=PTR_TO_CTX и insn имеет R2=R1+R1, то R2=SCALAR_VALUE, поскольку сложение двух действительных указателей даёт недействительный указатель (в защищённом (secure) режиме блок проверки будет отвергать любой тип арифметических операций с указателями, чтобы предотвратить утечку адресов из ядра к непривилегированным пользователям).

Если в регистр никогда не производилось записи, он будет нечитаемым. Программа

```
bpf_mov R0 = R2
bpf_exit
```

будет отвергнута, поскольку регистр R2 является нечитаемым при старте программы.

После вызова функции ядра, регистры R1 - R5 сбрасываются в нечитаемое состояние, а R0 имеет тип возврата из функции. Поскольку регистры R6-R9 сохраняет вызываемая функция, их состояние сохраняется в процессе вызова.

```
bpf_mov R6 = 1
bpf_call foo
bpf_mov R0 = R6
bpf_exit
```

[1]Directed acyclic graph.

Приведённая выше программа будет корректной. Если вместо R6 включить R1, программа будет отвергнута.

Команды загрузки и сохранения (load/store) разрешены только для регистров действительных типов, к которым относятся PTR_TO_CTX, PTR_TO_MAP, PTR_TO_STACK. Они ограничены и проверяются на предмет выравнивания. Например, программа

```
bpf_mov R1 = 1
bpf_mov R2 = 2
bpf_xadd *(u32 *)(R1 + 3) += R2
bpf_exit
```

будет отвергнута, поскольку R1 не имеет типа действительного указателя в момент выполнения bpf_xadd.

При старте R1 имеет тип PTR_TO_CTX (указатель на базовую struct bpf_context). Применяется обратный вызов (callback) для настройки блока проверки на разрешения программе eBPF доступа доступа лишь к некоторым полям структуры ctx с заданным размером и выравниванием. Например, insn.

```
bpf_ld R0 = *(u32 *)(R6 + 8)
```

намерена загрузить слово с адресом R6 + 8 и записи его в R0. Если R6=PTR_TO_CTX, блок проверки через обратный вызов is_valid_access() узнает, что по данные смещению 8 с размером 4 байта, доступны для чтения. В иных случаях блок проверки будет отвергать программу. Если R6=PTR_TO_STACK, доступ следует выровнять с сохранением в границах стека [-MAX_BPF_STACK, 0). В этом примере смещение равно 8, поэтому он не пройдёт проверку (выход за границу).

Блок проверки будет разрешать программе eBPF читать данные из стека лишь после того, как они будут записаны.

Блок проверки в традиционном BPF выполняет подобные операции со слотами памяти M[0-15]. Например, программа

```
bpf_ld R0 = *(u32 *)(R10 - 4)
bpf_exit
```

не будет корректной. Хотя R10 является корректным регистром, доступным лишь для чтения (read-only), и имеет тип PTR_TO_STACK, R10 - 4 находится в границах стека, в указанное ими место не было записи.

Заполнение регистров также отслеживается, поскольку сохраняемых вызываемой функцией 4 регистров (R6 - R9) для некоторых программ может быть недостаточно.

Разрешённые вызовы функций настраиваются с помощью bpf_verifier_ops->get_func_proto(). Блок проверки eBPF будет контролировать соблюдение требований по выравниванию для регистров. После вызова в регистре R0 устанавливается тип возврата из функции.

Вызов функции является основным механизмом расширения функциональности программ eBPF. Фильтры сокетов могут разрешать программам вызывать некий набор функций, а фильтры трассировки - совсем иной набор.

Если функция доступна программе eBPF, необходимо рассмотреть её с точки зрения безопасности. Блок проверки гарантирует вызов функции с действительными (пригодными) аргументами.

Фильтры seccomp и фильтры сокетов имеют разные ограничения по безопасности в традиционном BPF. Для seccomp эта задача решается двухэтапной проверкой и блок проверки традиционного BPF использует логику проверки seccomp. В eBPF для всех случаев применяется один настраиваемый блок проверки. Дополнительную информацию можно получить из файла kernel/bpf/verifier.c в дереве исходных кодов ядра.

Отслеживание значений регистров

Для определения безопасности программы eBPF блок проверки должен отслеживать диапазоны возможных значений каждого регистра и каждого гнезда (slot) в стеке. Это зачастую выполняется с помощью struct bpf_reg_state (определена в include/linux/bpf_verifier.h), обеспечивающей однотипное отслеживание для скаляров и указателей. Каждое состояние регистра имеет тип, который может быть NOT_INIT (в регистр не было записи), SCALAR_VALUE (некое значение, не являющееся указателем) или указатель. Типы указателей приведены в таблице.

PTR_TO_CTX	Указатель на bpf_context.
CONST_PTR_TO_MAP	Указатель на struct bpf_map. Это постоянная (const), поскольку арифметические операции с такими указателями запрещены.
PTR_TO_MAP_VALUE	Указатель на значение, сохранённое в элементе отображения (map).
PTR_TO_MAP_VALUE_OR_NULL	Указатель на значение отображения или NULL. Доступ к отображению (Отображения eBPF) возвращает этот тип, который становится PTR_TO_MAP_VALUE после проверки != NULL. Арифметические операции с такими указателями запрещены.
PTR_TO_STACK	Указатель на кадр.
PTR_TO_PACKET	skb->data.
PTR_TO_PACKET_END	skb->data + headlen, арифметические операции запрещены.
PTR_TO_SOCKET	Указатель на struct bpf_sock_ops с неявным учётом ссылок.
PTR_TO_SOCKET_OR_NULL	Указатель на сокет или NULL. Поиск сокета возвращает этот тип, который становится PTR_TO_SOCKET после проверки != NULL. Ссылки на PTR_TO_SOCKET учитываются, поэтому программы должны освобождать ссылки с помощью функции освобождения сокета перед завершением программы. Арифметические операции с такими указателями запрещены.

Однако указатель может быть смещён относительно этой базы (в результате арифметической операции), что отслеживается в двух частях - фиксированное и переменное смещение. Первая часть используется при добавлении к указателю точно известного значения (например, непосредственный операнд - imm), вторая - при добавлении неизвестных точно значений. Переменное смещение применяется также в SCALAR_VALUE для отслеживания диапазона возможных значений регистра.

Информация блока проверки (verifier) о переменном смещении состоит из нескольких частей, указанных ниже.

- Минимальное и максимальное значение в форме чисел без знака.
- Минимальное и максимальное значение в форме чисел со знаком.

- Сведения об отдельных битах в форме tnum - u64 mask и u64 value. Значение 1 в элементе mask представляет бит с неизвестным значением, 1 в value представляют биты со значением 1. Для битов со значением 0 указывается 0 в mask и value. Ни один из битов не может иметь значения 1 в маске и значении. Например, при чтении байта из памяти в регистр известно, что старшие 56 битов регистра имеют значение 0, а оставшиеся 8 неизвестны - это будет представляться в виде tnum (0x0; 0xff). Если затем для этого значения используется операция OR со значением 0x40, получается (0x40; 0xbf), а последующее сложение с 1 даёт (0x0; 0x1ff), из-за возможного переноса.

Помимо арифметических операций состояние регистра может меняться при ветвлении по условию. Например, при сравнении SCALAR_VALUE с условием > 8, ветвь true будет иметь значение (минимальное значение без знака) umin_value = 9, а ветвь false - umax_value = 8. Сравнение с учётом знака (BPF_JSGT или BPF_JSGE) будет обновлять минимальное и максимальное значение со знаком. Сведения о границах со знаком и без знака можно комбинировать, например, если сначала выполняется проверка value < 8, затем проверка s > 4, блок проверки может счесть, что value также > 4, а s< 8, поскольку границы не позволяют пересечь границу по знаку.

PTR_TO_PACKET с переменным смещением имеют идентификатор id, который является общим для всех указателей с тем же переменным смещением. Это важно для проверки диапазона в пакетах - после сложения переменной с регистром указателя для пакета (A) с последующим копированием в регистр B и прибавлением константы 4 к A оба регистра будут иметь общий идентификатор id, но A будет иметь фиксированное смещение +4. Если после этого для A выполняется проверка по границам и обнаруживается, что регистр меньше PTR_TO_PACKET_END, становится ясно, что регистр B имеет безопасный диапазон не менее 4 байтов. Диапазоны PTR_TO_PACKET более подробно описаны в параграфе Прямой доступ к пакетам.

Поле id применяется также в PTR_TO_MAP_VALUE_OR_NULL общем для всех указателей, возвращаемых при поиске в отображении (map). Это означает, что когда копия проверяется и обнаруживается, что она не пуста (non-NULL), все копии могут стать PTR_TO_MAP_VALUE. Как и проверка диапазонов, данные отслеживания служат для принудительного выравнивания доступа к указателям. Например, в большинстве систем указатель имеет размер 2 байта после выравнивания по 4-байтовой границе. Если программа добавляет 14 байтов для пропуска заголовка Ethernet, затем считывает поле IHL[1] и добавляет (IHL * 4), полученный в результате указатель будет иметь переменное смещение, известное как 4n+2 для некоторого n, поэтому добавление 2 байтов (NET_IP_ALIGN) обеспечивает 4-байтовое выравнивание, поэтому доступ к слову по этому указателю не создаёт опасности.

Поле id применяется также в PTR_TO_SOCKET и PTR_TO_SOCKET_OR_NULL, общих для всех копий указателя, возвращаемых при поиске сокета. Это похоже на обработку PTR_TO_MAP_VALUE_OR_NULL->PTR_TO_MAP_VALUE, но включает также отслеживание ссылок для указателя. PTR_TO_SOCKET неявно представляет ссылку на соответствующую struct sock. Для предотвращения утечки ссылки необходимо выполнить проверку на NULL и в случае non-NULL передать действительную ссылку функции освобождения сокета.

Прямой доступ к пакетам

В программах cls_bpf и act_bpf блок проверки разрешает прямой доступ к данным пакета по указателям skb->data и skb->data_end. Например,

```
1:  r4 = *(u32 *)(r1 +80)  /* загрузка skb->data_end */
2:  r3 = *(u32 *)(r1 +76)  /* загрузка skb->data */
3:  r5 = r3
4:  r5 += 14
5:  if r5 > r4 goto pc+16
R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt_end R5=pkt(id=0,off=14,r=14) R10=fp
6:  r0 = *(u16 *)(r3 +12) /* доступ к 12 и 13 байтам пакета */
```

Двухбайтовые загрузки (load) значений из пакета безопасны, поскольку программа выполняет проверку if (skb->data + 14 > skb->data_end) goto err в insn #5, в случае прохождения которой регистр R3 (указывает на skb->data) имеет по меньшей мере 14 доступных напрямую байтов. Блок проверки помечает его как R3=pkt(id=0,off=0,r=14). Здесь id=0 указывает отсутствие дополнительных переменных, прибавляемых к регистру, off=0 означает отсутствие прибавляемых констант, а r=14 указывает диапазон безопасного доступа, который говорит о доступности байтов [R3, R3 + 14).

Отметим, что регистр R5 указан как R5=pkt(id=0,off=14,r=14). Он тоже указывает данные пакета, но к регистру прибавляется константа 14 и он будет указывать skb->data + 14 и доступным диапазоном будет [R5, R5 + 14 - 14), содержащий 0 байтов.

Более сложный доступ к пакету может иметь вид

```
R0=inv1 R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt_end R5=pkt(id=0,off=14,r=14) R10=fp
6:  r0 = *(u8 *)(r3 +7)   /* загрузка 7-байта из пакета */
7:  r4 = *(u8 *)(r3 +12)
8:  r4 *= 14
9:  r3 = *(u32 *)(r1 +76) /* загрузка skb->data */
10:  r3 += r4
11:  r2 = r1
12:  r2 <<= 48
13:  r2 >>= 48
14:  r3 += r2
15:  r2 = r3
16:  r2 += 8
17:  r1 = *(u32 *)(r1 +80)       /* загрузка skb->data_end */
18:  if r2 > r1 goto pc+2
              R0=inv(id=0,umax_value=255,var_off=(0x0;    0xff))    R1=pkt_end    R2=pkt(id=2,off=8,r=8)
R3=pkt(id=2,off=0,r=8) R4=inv(id=0,umax_value=3570,var_off=(0x0; 0xfffe)) R5=pkt(id=0,off=14,r=14) R10=fp
19:  r1 = *(u8 *)(r3 +4)
```

Регистр R3 имеет состояние R3=pkt(id=2,off=0,r=8), id=2 означает, что видны две инструкции r3 += rX, поэтому R3 указывает некое смещение в пакете, а поскольку программа выполняет проверку if (r3 + 8 > r1) goto err в insn #18,

[1] IP header length - размер заголовка IP.

безопасным диапазоном будет [R3, R3 + 8). Блок проверки разрешает доступ лишь операциям сложения и вычитания (add'/'sub) к регистрам пакета. Все прочие операции будут устанавливать регистр в состояние SCALAR_VALUE и он будет недоступен для прямого доступа к пакету.

Операция r3 += rX может приводить к переполнению, когда результат станет меньше исходного skb->data и блок проверки должен предотвращать это. Поэтому при наличии команды r3 += rX со значением rX более 16 битов любая последующее сравнение границы r3 с skb->data_end не будет давать сведений о диапазоне и попытки чтения по этому указателю приведёт к ошибке (invalid access to packet). Например, после r4 = *(u8 *)(r3 +12) (insn #7 выше) регистр R4 будет иметь состояние R4=inv(id=0,umax_value=255,var_off=(0x0; 0xff)), которое означает, что 56 старших битов регистра гарантированно имеют значение 0, а о младших 8 битах ничего не известно. После r4 *= 14 состояние становится R4=inv(id=0,umax_value=3570,var_off=(0x0; 0xfffe)), так как умножение 8-битового значения на константу 14 сохранит в 52 старших битах значение 0, а младший бит будет иметь значение 0, поскольку число 14 чётное. Аналогично r2 >>= 48 будет давать R2=inv(id=0,umax_value=65535,var_off=(0x0; 0xffff)), поскольку сдвиг не является расширением знака. Эта логика реализована в функции adjust_reg_min_max_vals(), вызывающей adjust_ptr_min_max_vals() для сложения указателя со скаляром (или наоборот) и adjust_scalar_min_max_vals() для операции над двумя скалярами.

В конечном итоге программа BPF может получить доступ напрямую с использованием обычного кода C.

```
void *data = (void *)(long)skb->data;
void *data_end = (void *)(long)skb->data_end;
struct eth_hdr *eth = data;
struct iphdr *iph = data + sizeof(*eth);
struct udphdr *udp = data + sizeof(*eth) + sizeof(*iph);

if (data + sizeof(*eth) + sizeof(*iph) + sizeof(*udp) > data_end)
        return 0;
if (eth->h_proto != htons(ETH_P_IP))
        return 0;
if (iph->protocol != IPPROTO_UDP || iph->ihl != 5)
        return 0;
if (udp->dest == 53 || udp->source == 9)
        ...;
```

Это делает написание программы более простым по сравнению с LD_ABS insn и существенно ускоряет процесс.

Отображения eBPF

Отображение (map) является базовым хранилищем различных типов данных, совместно используемых ядром и пользовательским пространством. Доступ к map из пользовательского пространства выполняется с помощью системных вызовов BPF, указанных ниже.

- Для создания отображения с заданным типом и атрибутами служит функция map_fd = bpf(BPF_MAP_CREATE, union bpf_attr *attr, u32 size), использующая attr->map_type, attr->key_size, attr->value_size, attr->max_entries и возвращающая файловый дескриптор локального процесса или отрицательный код ошибки.
- Для поиска по ключу в заданном отображении применяется функция err = bpf(BPF_MAP_LOOKUP_ELEM, union bpf_attr *attr, u32 size), использующая attr->map_fd, attr->key, att->value и возвращающая 0 с сохранением найденного элемента или отрицательный код ошибки.
- Для создания и обновления пар (ключ, значение) в данном отображении служит функция err = bpf(BPF_MAP_UPDATE_ELEM, union bpf_attr *attr, u32 size), использующая attr->map_fd, attr->key, attr->value и возвращающая 0 или отрицательный код ошибки.
- Для поиска и удаления элемента по ключу в данном отображении служит функция err = bpf(BPF_MAP_DELETE_ELEM, union bpf_attr *attr, u32 size), использующая attr->map_fd, attr->key.
- Для удаления отображения служит close(fd).

При завершении процесса отображения удаляются автоматически. Программы пользовательского пространства применяют указанные вызовы для создания отображений и доступа к ним, а программы eBPF могут в то же время обновлять отображения. Отображения могут иметь тип hash, array, bloom filter, radix-tree и т. п. Отображение определяется:

- типом;
- максимальным числом элементов;
- размером ключа в байтах;
- размером значения в байтах.

Сокращение проверки

На практике блок проверки не проходит по всем возможным путям в программе. Для каждого анализируемого ветвления блок просматривает все состояния, в которых он был ранее при выполнении этой инструкции. Если какое-либо из них содержит текущее состояние как подмножество, ветвь «сокращается», т. е. факт предшествующего восприятия состояния подразумевает принятие текущего. Например, если в предыдущем состоянии R1 содержит указатель на пакет, а в текущем - указатель на пакет с таким же или большим размером и с не менее строгим выравниванием, R1 считается безопасным. Точно так же, если регистр R2 имел раньше состояние NOT_INIT, он не мог использоваться из этой точки, поэтому любое значение в R2 (включая NOT_INIT) безопасно. Реализация этого обеспечивается функцией regsafe(). Сокращение учитывает не только регистры, но и стек (и все регистры, которые он может включать). Все они должны быть безопасными для сокращения ветви. Это реализовано в функции states_equal().

Сообщения при проверке eBPF

Ниже приведено несколько примеров недействительных программ eBPF и сообщений блока проверки об ошибках, записываемых в журнал.

Программа с недоступными инструкциями

```
static struct bpf_insn prog[] = {
BPF_EXIT_INSN(),
BPF_EXIT_INSN(),
};
```

Ошибка

```
unreachable insn 1
```

Программа, читающая неинициализированный регистр

```
BPF_MOV64_REG(BPF_REG_0, BPF_REG_2),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (bf) r0 = r2
R2 !read_ok
```

Программа, не инициализирующая регистр R0 перед выходом

```
BPF_MOV64_REG(BPF_REG_2, BPF_REG_1),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (bf) r2 = r1
1: (95) exit
R0 !read_ok
```

Программа с доступом за границу стека

```
  BPF_ST_MEM(BPF_DW, BPF_REG_10, 8, 0),
  BPF_EXIT_INSN(),
```

Ошибка

```
  0: (7a) *(u64 *)(r10 +8) = 0
  invalid stack off=8 size=8
```

Программа, не инициализирующая стек до передачи его адреса функции

```
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (bf) r2 = r10
1: (07) r2 += -8
2: (b7) r1 = 0x0
3: (85) call 1
```

Недействительное непрямое чтение из стека по смещению -8+0 с размером 8.

Программа, использующая недействительный дескриптор map_fd=0 при вызове функции map_lookup_elem()

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 0x0
4: (85) call 1
```

fd 0 не указывает действительное отображение bpf_map.

Программа, не проверяющая возвращаемое map_lookup_elem() значение перед доступом к отображению

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 0),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 0x0
4: (85) call 1
5: (7a) *(u64 *)(r0 +0) = 0
```

R0 указывает недействительный доступ к памяти map_value_or_null.

Программа с корректной проверкой возврата из map_lookup_elem() значения NULL и доступом к памяти с некорректным аргументом

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 1),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 4, 0),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 1
4: (85) call 1
5: (15) if r0 == 0x0 goto pc+1
 R0=map_ptr R10=fp
6: (7a) *(u64 *)(r0 +4) = 0
```

Несогласованный доступ по смещению 4 с размером 8.

Программа с корректной проверкой возврата из map_lookup_elem() значения NULL и доступом к памяти с корректным аргументом на одной стороне ветвления if, но без этого на другой стороне if

```
BPF_ST_MEM(BPF_DW, BPF_REG_10, -8, 0),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_LD_MAP_FD(BPF_REG_1, 0),
BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 0),
BPF_EXIT_INSN(),
BPF_ST_MEM(BPF_DW, BPF_REG_0, 0, 1),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (7a) *(u64 *)(r10 -8) = 0
1: (bf) r2 = r10
2: (07) r2 += -8
3: (b7) r1 = 1
4: (85) call 1
5: (15) if r0 == 0x0 goto pc+2
 R0=map_ptr R10=fp
6: (7a) *(u64 *)(r0 +0) = 0
7: (95) exit
from 5 to 8: R0=imm0 R10=fp
8: (7a) *(u64 *)(r0 +0) = 1
R0 invalid mem access 'imm'
```

Программа, выполняющая поиск сокета, затем устанавливающая указатель NULL без проверки

```
BPF_MOV64_IMM(BPF_REG_2, 0),
BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_2, -8),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_MOV64_IMM(BPF_REG_3, 4),
BPF_MOV64_IMM(BPF_REG_4, 0),
BPF_MOV64_IMM(BPF_REG_5, 0),
BPF_EMIT_CALL(BPF_FUNC_sk_lookup_tcp),
BPF_MOV64_IMM(BPF_REG_0, 0),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (b7) r2 = 0
1: (63) *(u32 *)(r10 -8) = r2
2: (bf) r2 = r10
3: (07) r2 += -8
4: (b7) r3 = 4
5: (b7) r4 = 0
6: (b7) r5 = 0
7: (85) call bpf_sk_lookup_tcp#65
8: (b7) r0 = 0
9: (95) exit
```

Не освобождённая ссылка id=1, alloc_insn=7.

Программа, выполняющая поиск сокета, но не проверяющая возвращаемое значение на «не NULL»

```
BPF_MOV64_IMM(BPF_REG_2, 0),
BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_2, -8),
BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -8),
BPF_MOV64_IMM(BPF_REG_3, 4),
BPF_MOV64_IMM(BPF_REG_4, 0),
BPF_MOV64_IMM(BPF_REG_5, 0),
BPF_EMIT_CALL(BPF_FUNC_sk_lookup_tcp),
BPF_EXIT_INSN(),
```

Ошибка

```
0: (b7) r2 = 0
1: (63) *(u32 *)(r10 -8) = r2
2: (bf) r2 = r10
3: (07) r2 += -8
4: (b7) r3 = 4
5: (b7) r4 = 0
6: (b7) r5 = 0
7: (85) call bpf_sk_lookup_tcp#65
8: (95) exit
```

Не освобождённая ссылка id=1, alloc_insn=7.

Тестирование

Наряду с инструментами BPF ядро включает тестовый модуль с различными вариантами проверки для традиционного и внутреннего BPF, которые можно выполнить с интерпретатором BPF и компилятором JIT. Модуль размещается в файле lib/test_bpf.c и включается через Kconfig

```
CONFIG_TEST_BPF=m
```

После сборки и установки модуля тесты можно выполнить путём активизации модуля test_bpf с помощью insmod или modprobe. Результаты тестов, включая время в наносекундах, можно увидеть в журнале ядра с помощью dmesg.

Пакет trinity

Пакет trinity для тестирования системных вызовов Linux имеет встроенную поддержку BPF и SECCOMP-BPF.

Авторы

Этот документ был написан для того, чтобы дать заинтересованным читателям представление о базовой архитектуре.

- Jay Schulist <jschlst@samba.org>
- Daniel Borkmann <daniel@iogearbox.net>
- Alexei Starovoitov <ast@kernel.org>

Литература

[1] Documentation/userspace-api/seccomp_filter.rst (исходный код ядра).

[2] Steven McCanne, Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93). USENIX Association, Berkeley, CA, USA, 2-2. http://www.tcpdump.org/papers/bpf-usenix93.pdf.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru