BPF: Indirect Jumps

Anton Protopopov
ISOVALENT
now part of CISCO

LINUX
PLUMBERS
CONFERENCE Vienna, Austria / Sept. 18-20, 2024

Contents

- BPF Static Keys (in a Nutshell)
- Instruction Set Maps
- Indirect Branches in BPF

BPF Static Keys

```
__section("kprobe/__x64_sys_getpgid")
int worker(void *ctx)
{
        if (bpf_static_branch_unlikely(&debug_key))
                bpf_printk("__x64_sys_getpgid\n");
        return 0;
}
```

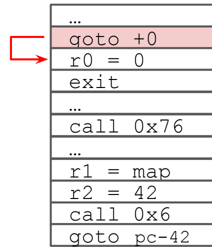
BPF Static Keys: branch is unlikely, key is off

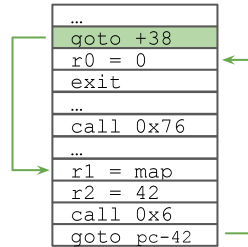
```
int worker(void * ctx):
; asm goto("1:"
  0: (05) goto pc+0
; return 0;
  1: (b7) r0 = 0
  2: (95) exit
; bpf_printk("__x64_sys_getpgid");
  3: (18) r1 = map[id:31][0]+0
  5: (b7) r2 = 18
  6: (85) call bpf_trace_printk#-79456
  7: (05) goto pc-7
```

BPF Static Keys: branch is unlikely, key is off

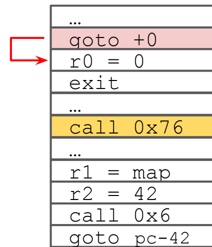
```
int worker(void * ctx):
; asm goto("1:"
   0: (05) goto pc+2
; return 0;
   1: (b7) r0 = 0
   2: (95) exit
; bpf_printk("__x64_sys_getpgid");
   3: (18) r1 = map[id:41][0]+0
   5: (b7) r2 = 18
   6: (85) call bpf_trace_printk#-79456
   7: (05) goto pc-7
```

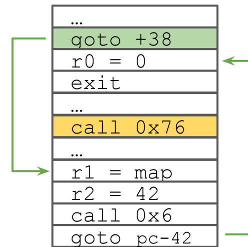
Static Keys vs. Relocations



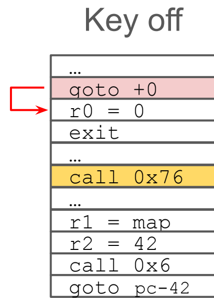


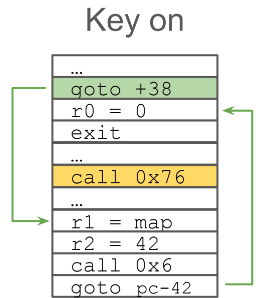
Static Keys vs. Relocations



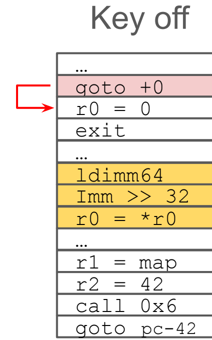


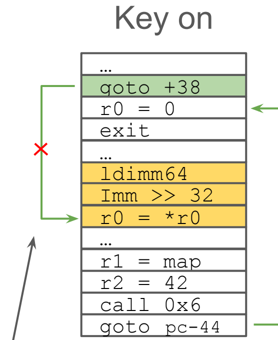
Static Keys vs. Relocations



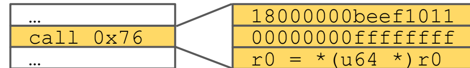


Load, verify, relocate



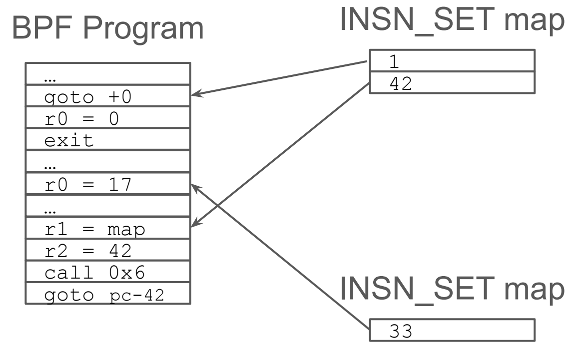


bpf_patch_insn_data()

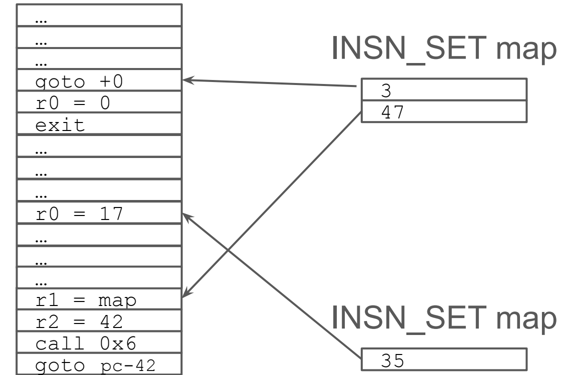


This jump becomes invalid, we need to relocate it the same way the normal jumps are relocated

A new BPF map: Instruction Set



Load, verify, relocate



Instruction Set Map Properties

Before program load

- A map is populated with instructions offsets

On program load:

- The map becomes read-only to userspace
  (it’s always read-only on the BPF side)
- Every instruction in this map is properly relocated

BPF Static Keys API

With such a map configuring a static key is easy:

```
bpf(STATIC_KEY_UPDATE,
    attrs={.key = map_fd, .on = <bool>})
```

* See more details on BPF Static Keys in [1] and [2]

Why new map: Instruction Set

- Jump offsets can be stored in 16-byte jumps (we’re adding new instructions in any case)
- However, this is more useful to have an object which groups to simplify static keys API
- What is even more important is that instruction sets can also be used for other purposes, e.g., to implement Indirect Branches

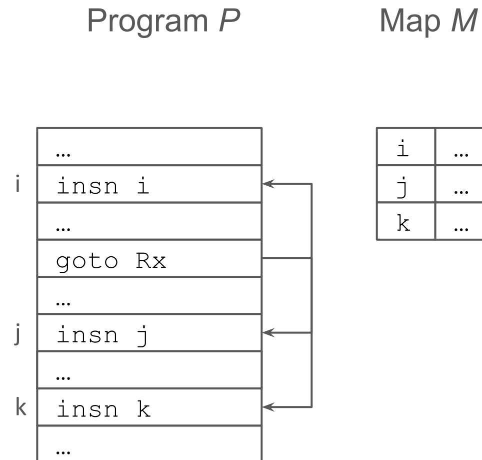
Indirect Jumps

- The goal is to add a new instruction (or multiple)

```
goto Rx
goto *Rx
(or so, see later)
```

Indirect Jumps: possible with Instruction Set Maps

A `goto rx` instruction must point to an instruction set map. Then during verification we can check that 1) Every jump from `goto rX` is valid 2) Rx is actually loaded from *M*

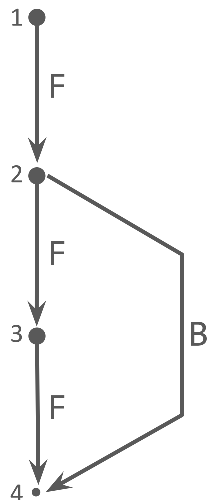


Indirect Jumps: new instructions

This looks reasonable to add the following instructions:

- BPF_JMP|BPF_JA|**BPF_X**, **src=index, dst=0, imm=map**
  - *Translated by the verifier to the second form*
- BPF_JMP|BPF_JA|**BPF_X**, **src=map_value, dst=1, imm=map**
  - *Jitted to, e.g., `jmpq *(%rsi)`*
- BPF_JMP|BPF_JA|**BPF_X**, **src=ip, dst=2, imm=map**
  - *Jitted to, e.g., `jmpq *%rsi`*
  - *(This one looks like an optional + requires more verification)*

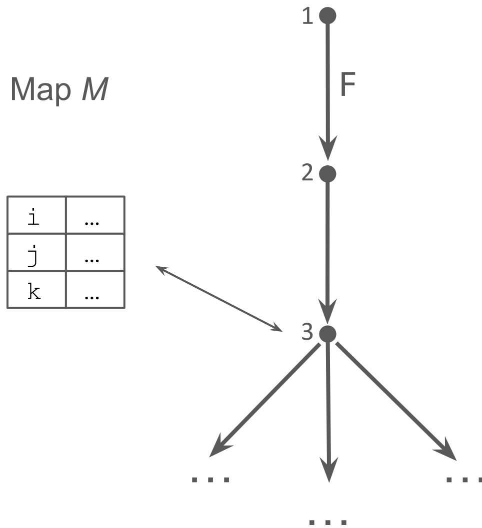
Orthodox Jumps: check_cfg & visit_insn



Vertex state (from here):

```
enum {
        DISCOVERED = 0x10,
        EXPLORED = 0x20,
        FALLTHROUGH = 1,
        BRANCH = 2,
};
```

Indirect Jumps: check_cfg & visit_insn



Vertex state:

- Old orthodox state
- + A new Counter inside insn_state

Entering goto rx instruction:

```
u16 last_edge = GET_HIGH(insn_state[t]);
if (last_edge == map->max_entries)
        return DONE_EXPLORING;
```

Pushing the next insns to stack:

```
w = insn_set_xlated_offset(map, last_edge);
SET_HIGH(insn_state[t], last_edge + 1);
insn_stack[env->cfg.cur_stack++] = w;
```

Indirect Jumps: Verification (do_check)

```
umin = regs[insn->src_reg].umin_value;
umax = regs[insn->src_reg].umax_value;

if (umax >= map->max_entries)
        return -EINVAL;

for (i = umin + 1; i < map->max_entries; i++) {
        idx = insn_set_xlated_offset(map, i);
        other_branch = push_stack(env, idx, env->insn_idx, false);
}

env->insn_idx = insn_set_xlated_offset(map, umin);
continue;
```

* Only DST=0 version of instruction is supported ATM. For DST=1 it must be checked that *src_reg* was loaded from the *map*

Indirect Jumps: prepare to JIT: *DST=0* -> *DST=1*

```
/*
 * Replace BPF_JMP|BPF_JA|BPF,SRC=Rx,DST=0,IMM=fd with
 *
 * Rt = ldimm64(map_address)
 * Rt += "offset to elements"
 * Rx *= element size
 * Rx += Rt
 * BPF_JMP|BPF_JA|BPF,SRC=Rx,DST=1,IMM=fd
 */

*patch++ = BPF_RAW_INSN(BPF_LD | BPF_IMM | BPF_DW, BPF_REG_AX, 0, 0, (u32)(u64)map);
*patch++ = BPF_RAW_INSN(0, 0, 0, 0, (u32)((u64)map >> 32));
*patch++ = BPF_ALU64_IMM(BPF_ADD, BPF_REG_AX, sizeof(struct bpf_map));
*patch++ = BPF_ALU64_IMM(BPF_MUL, insn->src_reg, sizeof(struct insn_ptr));
*patch++ = BPF_ALU64_REG(BPF_ADD, insn->src_reg, BPF_REG_AX);
*patch++ = BPF_RAW_INSN(BPF_JMP | BPF_JA | BPF_X, 1, insn->src_reg, 0, insn->imm);
```

Indirect Jumps: prepare to JIT: *DST=0* -> *DST=1*

```
/*
 * Replace BPF_JMP|BPF_JA|BPF,SRC=Rx,DST=0,IMM=fd with
 *
 * Rt = ldimm64(map_address)
 * Rt += "offset to elements"
 * Rx *= element size
 * Rx += Rt
 * BPF_JMP|BPF_JA|BPF,SRC=Rx,DST=1,IMM=fd
 */
```

```
struct insn_ptr {
        void *jitted_ip;
        u32 jitted_off;
        u32 jitted_len;
        int jitted_jump_offset;
        u32 xlated_off;
};
```

```
*patch++ = BPF_RAW_INSN(BPF_LD | BPF_IMM | BPF_DW, BPF_REG_AX, 0, 0, (u32)(u64)map);
*patch++ = BPF_RAW_INSN(0, 0, 0, 0, (u32)((u64)map >> 32));
*patch++ = BPF_ALU64_IMM(BPF_ADD, BPF_REG_AX, sizeof(struct bpf_map));
*patch++ = BPF_ALU64_IMM(BPF_MUL, insn->src_reg, sizeof(struct insn_ptr));
*patch++ = BPF_ALU64_REG(BPF_ADD, insn->src_reg, BPF_REG_AX);
*patch++ = BPF_RAW_INSN(BPF_JMP | BPF_JA | BPF_X, 1, insn->src_reg, 0, insn->imm);
```

Indirect Jumps: JIT

- The only change to [x86] Jit is to add the following case

```
case BPF_JMP | BPF_JA | BPF_X:
case BPF_JMP32 | BPF_JA | BPF_X:
        __emit_indirect_jump(&prog, insn->src_reg);
        break;
```

Code example 1

```
struct bpf_insn insns[] = {
        BPF_MOV64_IMM(BPF_REG_1, 0),
        BPF_GOTO_X(1),
        BPF_MOV64_IMM(BPF_REG_0, XDP_PASS),
        BPF_EXIT_INSN(),
};

__u32 offsets[] = { 2 };
int map_fd = _bpf_insn_set_create(offsets, 1);

insns[1].imm = map_fd;
```

Code example 1

```
0:   r1 = 0
1:   goto r1
2:   r0 = 2
3:   exit
```

```
struct bpf_insn insns[] = {
        BPF_MOV64_IMM(BPF_REG_1, 0),
        BPF_GOTO_X(1),
        BPF_MOV64_IMM(BPF_REG_0, XDP_PASS),
        BPF_EXIT_INSN(),
};

__u32 offsets[] = { 2 };
int map_fd = _bpf_insn_set_create(offsets, 1);

insns[1].imm = map_fd;
```

Create an instruction set map of size 1 with one value: M={2}

Put the map file descriptor inside the instruction. Now program is ready to load.

Code example 1

```
struct bpf_insn insns[] = {
        BPF_MOV64_IMM(BPF_REG_1, 0),
        BPF_GOTO_X(1),
        BPF_MOV64_IMM(BPF_REG_0, XDP_PASS),
        BPF_EXIT_INSN(),
};

__u32 offsets[] = { 2 };
int map_fd = _bpf_insn_set_create(offsets, 1);

insns[1].imm = map_fd;
```

```
# bpftool p dump x id 18
   0: (b7) r1 = 0
   1: (18) r11 = map[id:4]
   3: (07) r11 += 248
   4: (27) r1 *= 24
   5: (0f) r1 += r11
   6: (0d) goto *(r1)
   7: (b7) r0 = 2
   8: (95) exit
```

Code example 2

```
        __u8 _insns[] = {
/*start:*/
                0x85, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, // call 0x7
                0xbf, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // r6 = r0
/*repeat:*/
                0xbf, 0x62, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // r2 = r6
                0x57, 0x02, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, // r2 &= 0x1
                0x0d, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // goto r2
/*null:*/
                0x85, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, // call 0x7
                0x05, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, // goto +0x2 <cont>

/*eis:*/
                0x85, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, // call 0x7
                0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // goto +0x0 <cont>
/*cont:*/
                0x15, 0x06, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, // if r6 == 0x0 goto +0x3 <end>
                0x77, 0x06, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, // r6 >>= 0x1
                0xe5, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, // may_goto +1
                0x05, 0x00, 0xf5, 0xff, 0x00, 0x00, 0x00, 0x00, // goto -0xb <repeat>

/*end:*/
                0xb7, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, // r0 = 0x2
                0x95, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // exit
        };
```

C jump tables

```
void *x[] = { &&_x, &&_y, &&_z };

if (i < 2)
        goto *x[i];
```

C jump tables

```c
void *x[] = { &&_x, &&_y, &&_z };

if (i < 2)
        goto *x[i];
```

```llvm
%4 = getelementptr inbounds nuw [3 x ptr], ptr @x, i64 0, i64 %0
%5 = load ptr, ptr %4, align 8, !dbg !31, !tbaa !32
indirectbr ptr %5, [label %8, label %6, label %7]
```

C jump tables

```
%4 = getelementptr inbounds nuw [3 x ptr], ptr @x, i64 0, i64 %0
%5 = load ptr, ptr %4, align 8, !dbg !31, !tbaa !32
indirectbr ptr %5, [label %8, label %6, label %7]
```

```
r1 <<= 3

r2 = .L__const.foo.x ll
r2 += r1
r1 = *(u64 *)(r2 + 0)

goto *(r1)
```

C jump tables

llvm optimizer

```
r1 <<= 3

r2 = .L__const.foo.x ll
r2 += r1
r1 = *(u64 *)(r2 + 0)

goto *(r1)
```

```
# r1 contains index
r2 = .L__const.foo.x ll
goto *(r1)
```

libbpf

JMP|JA|X, .src=r1, .imm=map(x)

C Switches

- All the switches are now replaced with if-else-...
- Larger switches can benefit from using an indirect jump
- Yonghong said:
  - First support for jump tables
  - Then support for switches

Next Steps

- Start pushing the patch set: insn set map, then static keys

- Indirect branches:
  1. LLVM code generation + libbpf support (jump_tables)
  2. Refactor the kernel side accordingly
  3. may_goto 1

- C Switches => indirect jumps

Questions?