Compile once, run everywhere

Nov 2018

Yonghong Song and Alexei Starovoitov

facebook

Motivation

tracing bpf programs that look into guts of the kernel require on the fly compilation of every server. BCC embeds clang and llvm for this purpose. This is problematic for several reasons

- kernel-devel package needs to be installed on every host, since clang needs to process kernel headers to figure out struct field offsets
- kernel internal headers are not part of kernel-devel and tracing programs cannot access structs defined in headers like btrfs_inode.h. Current workaround is to copy-paste struct definitions into bpf program which is suboptimal
- hand built debug kernels (for installing via tar-pkg on test machines or for kvm runs) don't have headers, so all of bcc-based tools don't work
- libbcc.so size is 60Mbyte. 99% of it is clang/llvm with large .text section that consumes memory after BCC compiled and loaded bpf programs. BCC frees most of run-time heap memory, but cannot free it all and cannot free .text

Introduce 'field offset relocations' BTF section

```
r6 = *(u64 *)(r1 + 112)
r6 += 1176
r7 = r10
r7 += -4
r1 = r7
r2 = 4
r3 = r6
call 4
r1 = r7
r2 = 4
r3 = r6
call 4
r0 = 0
exit
```

relocation:

- BTF type_id of 'struct task_struct'
- name of the field 'pid'
- BTF type_id of the field 'pid'

```
SEC("kprobe/__set_task_comm")
int prog(struct pt_regs *ctx)
{
        struct task_struct *tsk;
        typeof(tsk->pid) pid;

        tsk = PT_REGS_PARM1(ctx);

        bpf_probe_read(&pid, sizeof(pid), &tsk->pid);
        bpf_probe_read(&pid, sizeof(pid), (void *)tsk + __builtin_offsetof(struct task_struct, pid));
        return 0;
}
```

Ongoing work

- modify clang, llvm to emit relocations into .o
- modify libbpf to apply relocations to BPF instructions in user space prior to loading
- read vmlinux and kernel modules dwarf, generate corresponding BTF, embed it into vmlinux and .ko
  - vmlinux: from 120 Mbyte of dwarf types to 2 Mbytes of BTF

Issues: #defines (not present in LLVM IR)

```
struct tcpbpf_minmax_sample {
  u32 v1;
  u32 v2;
};
u32 minrtt;
bpf_probe_read(&ms, sizeeof(ms), tp->rtt_min);
#ifdef KERNEL_VERSON  == 406
  minrtt = ms.v1;
#else
  minrtt = ms.v2;
#endif
…
```

- replace macro with global variable, e.g., __bpf_kernel_version
- compile all branches, in libbpf, user need to replace __bpf_kernel_version with constant w.r.t the kernel on the host.

```
…
if (__bpf_kernel_verson == 406)
  minrtt = ms.v1;
eelse
  minrtt = ms.v2;
…
```

Issues: harder #defines

```
#ifdef REQ_WRITE
    data.rwflag = !!(req->cmd_flags & REQ_WRITE);
#elif defined(REQ_OP_SHIFT)
    data.rwflag = !!((req->cmd_flags >> REQ_OP_SHIFT) == REQ_OP_WRITE);
#else
    data.rwflag = !!((req->cmd_flags & REQ_OP_MASK) == REQ_OP_WRITE);
#endif
```

- Still use global variable approach, where actual macro value is patched by libbpf

```
if (__bpf_req_write)
  data.rwflag = !!(req->cmd_flags & __bpf_req_write);
else if (__bpf_req_op_shift)
  data.rwflag = !!((req->cmd_flags >> __bpf_req_op_shift) == REQ_OP_WRITE);
else
  data.rwflag = !!((req->cmd_flags & REQ_OP_MASK) == REQ_OP_WRITE);
```

Issues: #defines with unknown types

```
   // Get port
   evt.lport = inet->inet_sport;
   evt.lport = ntohs(evt.lport);

   // Get network namespace id, if kernel supports it
#ifdef CONFIG_NET_NS
   evt.netns = sk->__sk_common.skc_net.net->ns.inum;
#else
   evt.netns = 0;
#endif
```

bcc style

```
typedef struct {
#ifdef CONFIG_NET_NS
      struct net *net;
#endif
} possible_net_t;
```

```
#ifdef CONFIG_NET_NS
  struct net *net;
  bpf_probe_read(&net, sizeof(struct net *), &sk->__sk_common.skc_net.net);
  bpf_probe_read(&evt.netns, sizeof(evt.netns), &net->ns.inum);
#else
  evt.netns = 0;
#endif
```

Solution #1: fuzzy struct, fuzzy type matching

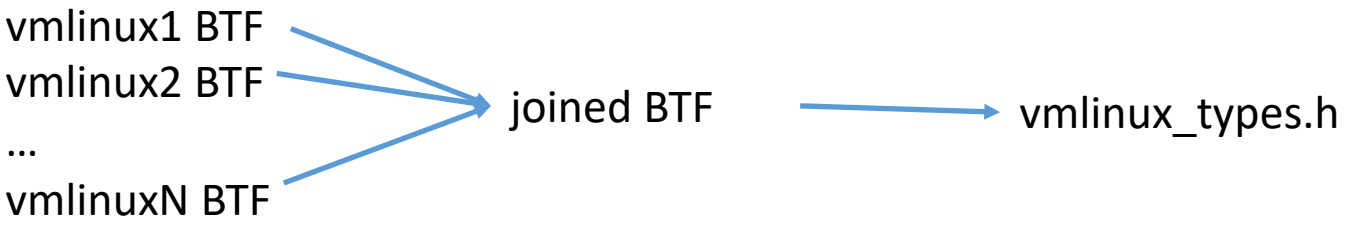
```
if (__bpf_config_net_ns) {
  struct __bpf_possible_net_t { struct __bpf_net *net; }
  struct __bpf_net { struct __bpf_ns_common ns; }
  struct __bpf_ns_common {unsigned int inum;}

  struct __bpf_net *net;
  bpf_probe_read(&net, sizeof(struct __bpf_net *),
    &((struct __bpf_possible_net_t *)&sk->__sk_common.skc_net)->net);
  bpf_probe_read(&evt.netns, sizeof(evt.netns), &net->ns.inum);
}
```

Fuzzy struct, one member per struct

- libbpf loader do fuzzy matching vs. vmlinux BTF
- Predefined macros can help but still need user involvement
- Possible compiler automation

Solution #2: Joining vmlinux BTFs




```
#include “vmlinux_types.h”

if (__bpf_config_net_ns) {
  evt.netns = sk->__sk_common.skc_net.net->ns.inum;
}
```

- Issues:
  - Same member with different type -> drop in joined BTF ?
  - Multiple (struct) types with the same name -> add version prefix ?

Other #define complications

- static inline functions -> copy into the bpf program?
- Macros - copy into the bpf program?
- Bigger code size? Less optimized code? Breaking verifier?

Introduce global variables and read only sections

```c
struct my_config {
        int max;
        int min;
} cfg;

int stats;

int bpf_prog(struct __sk_buff *skb)
{
        if (skb->len < cfg.min || skb->len > cfg.max) {
                stats++;
                bpf_printk("len %d\n", skb->len);
                return 1;
        }
        return 0;
}
```

```
        .text
bpf_prog:
        r3 = cfg                        # relocation into .bss
        r2 = *(u32 *)(r1 + 0)
        r4 = *(u32 *)(r3 + 4)
        if r4 > r2 goto LBB0_2
        r0 = 0
        r3 = *(u32 *)(r3 + 0)
        if r3 >= r2 goto LBB0_3
LBB0_2:
        r2 = stats                      # relocation into .bss
        r3 = *(u32 *)(r2 + 0)
        r3 += 1
        *(u32 *)(r2 + 0) = r3
        r2 = *(u32 *)(r1 + 0)
        r1 = .L.str                     # relocation into .rodata
        call 123#bpf_printk
        r0 = 1
LBB0_3:
        exit

        .comm   cfg,8,4                         # @cfg
        .comm   stats,4,4                       # @stats
        .section        .rodata.str1.1,"aMS",@progbits,1
.L.str:                                         # @.str
        .asciz  "len %d\n"
```

- no compiler changes
- proposed kernel API:
- globals – new map type with mandatory BTF
    - lookup/update syscalls by offset or by name
- rodata – another map type
    - single update() syscall before prog load (verifier will check null-termination and size of strings)