

Kernel func tracing in the face of compiler optimization

Yonghong Song and Alan Maguire

Overview

- Overview of kernel func tracing
- Compilation impacting kernel func tracing
 - Common with both clang and gcc
 - Clang optimizations
 - Gcc optimizations
- How to automate those special func tracing or make tracing those functions easier

Overview of Kernel Func Tracing

- Kprobe (BPF_PROG_TYPE_KPROBE)
 - tools/testing/selftests/bpf/progs/loop6.c
 - SEC("kprobe/virtqueue_add_sgs")
 - int BPF_KPROBE(trace_virtqueue_add_sgs, void *unused, struct scatterlist **sgs, unsigned int out_sgs, unsigned int in_sgs)
- fentry/fexit (BPF_PROG_TYPE_TRACING)
 - attach_type: BPF_TRACE_FENTRY, BPF_TRACE_EXIT
 - tools/testing/selftests/bpf/progs/test_d_path.c
 - SEC("fentry/flip_close")
 - int BPF_PROG(prog_close, struct file *file, void *id)
 - tools/testing/selftests/bpf/progs/socket_cookie_prog.c
 - SEC("fexit/inet_stream_connect")
 - int BPF_PROG(update_cookie_tracing, struct socket *sock, struct sockaddr *uaddr, int addr_len, int flags)
- The above BPF_KPROBE/BPF_PROG signatures are from kernel source.

Difficult Cases in Kernel Func Tracing

- Inlining
- Explicit function name changes (func.<suffixes>)
- Implicit function signature changes (same name)
- Implicit inlining
- Same static func/var names

Common Optimization with clang and gcc

- Complete inlining, the function is gone

```
static void * __bpf_copy_key(bpfptr_t ukey, u64 key_size)
{
    if (key_size)
        return kvmemdup_bpfptr(ukey, key_size);
    if (!bpfptr_is_null(ukey))
        return ERR_PTR(-EINVAL);
    return NULL;
}
$ grep __bpf_copy_key syscall.c
static void * __bpf_copy_key(bpfptr_t ukey, u64 key_size)
    key = __bpf_copy_key(ukey, map->key_size); // map_update_elem
    key = __bpf_copy_key(ukey, map->key_size); // map_delete_elem

$ grep __bpf_copy_key System.map
$
```

Tracing Complete Inlining Funcs

- llvm-dwarfdump vmlinux
- llvm-objdump -S vmlinux
- Kprobe + offset

```
static int map_update_elem(union bpf_attr *attr, bpfptr_t uattr)
{
    ...
    key = __bpf_copy_key(ukey, map->key_size);
    ...
}
static int __sys_bpf(int cmd, bpfptr_t uattr, unsigned int size)
{
    ...
    err = map_update_elem(&attr, uattr);
    ...
}
SYSCALL_DEFINE3(bpf, int, cmd, union bpf_attr __user *, uattr,
unsigned int, size)
{
    return __sys_bpf(cmd, USER_BPFPTR(uattr), size);
}
```

Tracing Complete Inlining Funcs

```
DW_TAG_inlined_subroutine
DW_AT_abstract_origin (0x022197c9 "bpf_copy_key")
DW_AT_entry_pc (0xffffffff8159a152)
DW_AT_GNU_entry_view (0x0000)
DW_AT_ranges (0x0012f585
[0xffffffff8159a152, 0xffffffff8159a169)
  [0xffffffff8159a169, 0xffffffff8159a170)
  ...)
DW_AT_call_file
("/home/yhs/work/bpf-next/kernel/bpf/syscall.c")
DW_AT_call_line (1693)
DW_AT_call_column (8)
DW_AT_sibling (0x02206daf)
DW_TAG_formal_parameter
DW_AT_abstract_origin (0x022197e8 "key_size")
DW_AT_location (0x00631ef6:
[0xffffffff8159a152, 0xffffffff8159a169): DW_OP_reg9 R9
  [0xffffffff8159ac74, 0xffffffff8159ac8a): DW_OP_reg9 R9
  ...
DW_TAG_formal_parameter
DW_AT_abstract_origin (0x022197db "ukey")
DW_AT_location ...
```

```
ffffffff81598050 <__sys_bpf>:
: {
ffffffff81598050: e8 bb 94 b7 ff    callq 0xffffffff81111510 <__fentry__>
ffffffff81598055: 55                pushq %rbp
...
;   key = bpf_copy_key(ukey, map->key_size);
ffffffff8159a120: 49 8d 7f 1c      leaq 0x1c(%r15), %rdi
ffffffff8159a124: 48 b8 00 00 00 00 fc ff df movabsq $-0x2000040000000000, %rax
# imm = 0xDFFFFFFC0000000000
ffffffff8159a12e: 48 89 fa        movq %rdi, %rdx
ffffffff8159a131: 48 c1 ea 03      shrq $0x3, %rdx
ffffffff8159a135: 0f b6 14 02      movzbl (%rdx,%rax), %edx
ffffffff8159a139: 48 89 f8        movq %rdi, %rax
ffffffff8159a13c: 83 e0 07        andl $0x7, %eax
ffffffff8159a13f: 83 c0 03        addl $0x3, %eax
ffffffff8159a142: 38 d0          cmpb %dl, %al
ffffffff8159a144: 7c 08          jl 0xffffffff8159a14e <__sys_bpf+0x20fe>
ffffffff8159a146: 84 d2          testb %dl, %dl
ffffffff8159a148: 0f 85 82 1f 00 00 jne 0xffffffff8159c0d0 <__sys_bpf+0x4080>
ffffffff8159a14e: 45 8b 4f 1c      movl 0x1c(%r15), %r9d
;   if (key_size)
ffffffff8159a152: 4d 85 c9        testq %r9, %r9
ffffffff8159a155: 0f 85 19 0b 00 00 jne 0xffffffff8159ac74 <__sys_bpf+0x2c24>
```

Selective Inlining

- Some call sites are inlined and others not inlined
- Symbol exists in the symbol table

```
// kernel/capability.c
bool ns_capable(struct user_namespace *ns, int cap)
{
    return ns_capable_common(ns, cap, CAP_OPT_NONE);
}

[$ ~/work/bpf-next/kernel (bpf)]$ grep " ns_capable(" *.c
capability.c:bool ns_capable(struct user_namespace *ns, int cap)
capability.c: return ns_capable(&init_user_ns, cap);
capability.c: return ns_capable(ns, cap) &&
ptrace.c: return ns_capable(ns, CAP_SYS_PTRACE);
signal.c: ns_capable(tcred->user_ns, CAP_KILL);
[$ ~/work/bpf-next/kernel (bpf)]$
/* other directories */
```

Solution:

- Non-LTO kernel:
 - Trace ns_capable()
 - Check whether any inline of ns_capable() in capability.c. If yes, use kprobe+offset to trace other inlined ns_capable() functions.
- LTO kernel:
 - Cross-file checking inlining of ns_capable().

Function Hot/Cold splits

- .cold suffix (both gcc and clang)
- For clang: -fsplit-machine-functions and need profile feedback data (<https://lore.kernel.org/all/20240728203001.2551083-6-xur@google.com/>)

```
[$ ~/work/bpf-next (master)]$ grep "\.cold" System.map | wc -l
39
[$ ~/work/bpf-next (master)]$ grep "\.cold" System.map
ffffff81ea8df9 t fpu_xstate_prctl.cold
ffffff81ea8e17 t __do_sys_kcmp.cold
ffffff81ea8e23 t second_overflow.cold
ffffff81ea8e2b t cgroup2_parse_param.cold
ffffff81ea8e35 t cgroup1_parse_param.cold
ffffff81ea8e3c t audit_comparator.cold
ffffff81ea8e3f t ftrace_match.cold
ffffff81ea8e46 t filter_match_preds.cold
ffffff81ea8e4d t process_fetch_insn.cold
ffffff81ea8e58 t process_fetch_insn.cold
ffffff81ea8e63 t process_fetch_insn.cold
ffffff81ea8e6e t trace_uprobe_register.cold
...
```

Function Hot/Cold Splits

```
Gcc 11.4.1
ffffff811dc430 <states_equal>:
;{
ffffff811dc430: e8 fb 1d e8 ff    callq 0xfffffff8105e230 <__fentry__>
ffffff811dc435: 41 57                pushq  %r15
...
;      if (i % BPF_REG_SIZE != BPF_REG_SIZE - 1)
ffffff811dc6a6: 83 f9 07            cmpl  $0x7, %ecx
ffffff811dc6a9: 75 53              jne   0xfffffff811dc6fe <states_equal+0x2ce>
;      switch (old->stack[spi].slot_type[BPF_REG_SIZE - 1]) {
ffffff811dc6ab: 41 80 7f 7f 05     cmpb  $0x5, 0x7f(%r15)
ffffff811dc6b0: 0f 87 cb c7 cc 00  ja   0xfffffff81ea8e81 <states_equal.cold>
ffffff811dc6b6: 41 0f b6 47 7f     movzbl 0x7f(%r15), %eax
ffffff811dc6bb: ff 24 c5 b0 7d 25 82  jmpq  *-0x7dda8250(,%rax,8)
...
ffffff81ea8e81 <states_equal.cold>:
;      switch (old->stack[spi].slot_type[BPF_REG_SIZE - 1]) {
ffffff81ea8e81: 45 31 e4          xorl  %r12d, %r12d
ffffff81ea8e84: e9 c5 35 33 ff     jmp   0xfffffff811dc44e <states_equal+0x1e>
```

All of following commands work fine:

```
echo 'p:states_equal states_equal+5 env=%ax'
> /sys/kernel/debug/tracing/kprobe_events
echo 'p:states_equal states_equal.cold+0
env=%ax' >
/sys/kernel/debug/tracing/kprobe_events
echo 'p:states_equal states_equal+0xccca51
env=%ax' >
/sys/kernel/debug/tracing/kprobe_events
echo 'p:states_equal 0xfffffff811dc430
env=%ax' >
/sys/kernel/debug/tracing/kprobe_events
echo 'p:states_equal 0xfffffff81ea8e81
env=%ax' >
/sys/kernel/debug/tracing/kprobe_events
```

Clang: Dead Argument Elimination

- Three instances in kernel/bpf/syscall.c: map_update_elem, map_delete_elem and strncpy_from_bpfptr.

```
typedef struct {  
    union {  
        void *kernel;  
        void __user *user;  
    };  
    bool is_kernel : 1;  
} sockptr_t;  
typedef sockptr_t bpfptr_t;  
static int map_update_elem(union bpf_attr *attr, bpfptr_t uattr) { ... }
```

LLVM IR before DeadArgElim:

```
i32 @map_update_elem(ptr noundef %attr, ptr nocapture readonly %uattr.coerce0, i8 %uattr.coerce1)
```

LLVM IR after DeadArgElim:

```
define internal fastcc i32 @map_update_elem(ptr noundef %attr, i8 %uattr.coerce1)
```

Clang: Dead Argument Elimination

```
/* include/linux/bpfptr.h */
static inline long strncpy_from_bpfptr(char *dst, bpfptr_t src, size_t count)
{
    if (bpfptr_is_kernel(src))
        return strncpy_from_kernel_nofault(dst, src.kernel, count);
    return strncpy_from_user(dst, src.user, count);
}
/* kernel/bpf/syscall.c */
static int bpf_prog_load(union bpf_attr *attr, bpfptr_t uattr, u32 uattr_size)
{
    ...
    if (strncpy_from_bpfptr(license,
        make_bpfptr(attr->license, uattr.is_kernel),
        sizeof(license) - 1) < 0)
        goto free_prog;
    ...
}

IR before: i64 @strncpy_from_bpfptr(ptr noundef %dst, ptr %src.coerce0, i8 %src.coerce1,
i64 noundef %count)
IR after: i64 @strncpy_from_bpfptr(ptr noundef %dst, ptr %src.coerce0, i8 %src.coerce1)
```

- strncpy_from_bpfptr() not inlined but become a static function.
- The third argument is a constant

Clang: Dead Argument Elimination

- Function name is the same after eliminating arguments!
- Current solution:
 - Check llvm IR or assembly code, extremely user unfriendly.
 - **Check dwarf callsite information, may not have enough info.**

```
0x000277ba: DW_TAG_subprogram
    DW_AT_low_pc   (0x000000000000b980)
    DW_AT_high_pc  (0x000000000000bfb9)
    DW_AT_frame_base (DW_OP_reg7 RSP)
    DW_AT_call_all_calls (true)
    DW_AT_name     ("map_update_elem")
```

...

```
0x000277ce: DW_TAG_formal_parameter
    DW_AT_name ("attr")
    DW_AT_type (0x00000241 "bpf_attr *")
0x000277da: DW_TAG_formal_parameter
    DW_AT_name ("uattr")
    DW_AT_type (0x0001a395 "bpfptr_t")
```

LLVM IR before DeadArgElim:

```
i32 @map_update_elem(ptr noundef %attr, ptr nocapture readonly
%uattr.coerce0, i8 %uattr.coerce1)
```

LLVM IR after DeadArgElim:

```
define internal fastcc i32 @map_update_elem(ptr noundef %attr, i8
%uattr.coerce1)
```

Clang: Dead Argument Elimination

- Possible solutions:
 - Similar to gcc, add proper suffixes.
 - Checking dwarf. If not enough, check IR or below remarks.
 - Add related transformations to llvm remarks with compilation flag `-foptimization-record-file=<file>`. Remarks giving detailed transformation information.
- Upstream Discussion
 - Current clang does not change function name even when function signature gets changed.
 - Add suffix:
<https://github.com/llvm/llvm-project/pull/105742>
 - Proposed: `.argelim`, `.retelim`

```
Pass:      deadargelim
Name:      ArgumentRemoved
DebugLoc:  { File: 'bpf-next/kernel/bpf/syscall.c', Line: 1670,
Column: 0 }
Function:  map_delete_elem
Args:
- String:  'eliminating argument '
- ArgName: uattr.coerce0
- String:  '('
- ArgIndex: '1'
- String:  ')'

Pass:      argpromotion
Name:      ArgumentPromoted
DebugLoc:  { File: 'bpf-next/net/mptcp/protocol.h', Line: 570,
Column: 0 }
Function:  mptcp_subflow_ctx
Args:
- String:  'promoting argument '
- ArgName: sk
- String:  '('
- ArgIndex: '0'
- String:  ')'
- String:  ' to pass by value'
```

Clang: Argument Promotion

tools/lib/subcmd/subcmd-util.h:

```
static inline void report(const char *prefix, const char *err, va_list params) {  
    char msg[1024];  
    vsnprintf(msg, sizeof(msg), err, params);  
    fprintf(stderr, "%s%s\n", prefix, msg);  
}
```

```
static __noreturn inline void die(const char *err, ...) {  
    va_list params;  
    va_start(params, err);  
    report(" Fatal: ", err, params);  
    exit(128);  
    va_end(params);  
}
```

tools/lib/subcmd/exec-cmd.c:

```
static const char *make_nonrelative_path(char *buf, size_t sz, const char *path) {  
    ...  
    die("Too long path: %.*s", 60, path);  
    ...  
}
```

IR before: void @report(ptr nocapture noundef readnone %prefix, ptr nocapture noundef
readonly %err, ptr noundef %params)

IR after: void @report(ptr nocapture noundef readonly %err, ptr noundef %params)

- O3 or LTO
- Func name remains the same.
- Actual func signature from `report(prefix, err, params)` to `report(err, params)`.

Clang: Argument Promotion

- Similar issue to Dead Argument Elimination
- Need suffix to identify signature change.
 - Proposed suffix: `.argprom`
- Remarks to give details of signature change. Alternatively, inspecting IR or assembly code.

Clang: ThinLTO

- Cross file inlining: `.llvm.<hash>` for both static functions and static variables
- **No signature change during promotion from static to global.**
- A few bpf related `.llvm.<hash>` symbols:
 - `Bpf_prog_show_fdinfo.llvm.13082368870908236650` (static func)
 - `Bpf_prog_release.llvm.13082368870908236650` (static func)
 - `fake_dst_ops.llvm.54750082607048300` (static var)
 - Libbpf handling `<symbol>.llvm.<hash>` properly when `<symbol>` is used in bpf program as a ksym.

(<https://lore.kernel.org/r/20240326041458.1198161-1-yonghong.song@linux.dev>)

Clang: FullLTO

```
ffffff8738a1f0 t create_dev.264
ffffff8738b330 t error.345
ffffff8738c760 t parse_header.196070
ffffff873bc810 t msr_init.10127
ffffff874127c0 t early_alloc.48669
ffffff8744aa20 t phy_init.129445
ffffff8745d7d0 t hid_init.159311
ffffff87464470 t bpf_iter_register.171711
ffffff87467120 t bpf_iter_register.175647
ffffff874775b0 t nofill.195996
ffffff87478210 t nofill.196042
ffffff879ce310 t phy_exit.129317
ffffff879cf4c0 t exit_rc_map.147977
ffffff879cfcc0 t exit_rc_map_pixelview.148935
ffffff879cfd00 t exit_rc_map_pixelview.148968
ffffff879cfec0 t exit_rc_map.149180
ffffff879cfee0 t exit_rc_map.149197
```

...

- <func>.<num> format
- **No signature when promoting functions to globals by adding suffix.**
- Bpf ksym like <func/var> will not work across different kernels if actual sym is <sym>.<num>. Libbpf didn't support this since FullLTO is not used commonly in kernel build.

Clang: Implicit Inlining

```
$ cat t.c
__attribute__((noinline)) int foo(void) { return 5; }
int bar(void) { return foo(); }
$ clang -O2 -c t.c
$ llvmoobjdump --no-show-raw-insn -d t.o
```

```
0000000000000000 <foo>:
   0:   movl  $0x5, %eax
   5:   retq
   6:   nopw  %cs:(%rax,%rax)
0000000000000010 <bar>:
  10:   movl  $0x5, %eax
  15:   retq
```

Implicitly inlined

```
$ cat t.c
__attribute__((noinline)) int foo(int v) { return v; }
int bar(void) { return foo(5); }
$ clang -O2 -c t.c
$ llvmoobjdump --no-show-raw-insn -d t.o
```

```
0000000000000000 <foo>:
   0:   movl  %edi, %eax
   2:   retq
   3:   nopw  %cs:(%rax,%rax)
0000000000000010 <bar>:
  10:   movl  $0x5, %eax
  15:   retq
```

Implicitly inlined

```
$ cat t.c
__attribute__((noinline)) int foo(int v) { return v + 1; }
int bar(void) { return foo(5); }
$ clang -O2 -c t.c
$ llvmoobjdump --no-show-raw-insn -d t.o
```

```
0000000000000000 <foo>:
   0:   leal  0x1(%rdi), %eax
   3:   retq
   4:   nopw  %cs:(%rax,%rax)
0000000000000010 <bar>:
  10:   movl  $0x5, %edi
  15:   jmp  0x1a <bar+0xa>
```

No inline

Gcc does not have this issue.

Clang: Implicit Inlining

```
$ cat t.c
#define __sink(expr) asm volatile("" : "+g"(expr))
__attribute__((noinline)) int foo(int v) { __sink(v); return v; }
int bar(void) { return foo(5); }
$ clang -O2 -c t.c
$ llvm-objdump --no-show-raw-insn -d t.o
```

```
0000000000000000 <foo>:
   0:   movl   %edi, %eax
   2:   movl   %edi, -0x4(%rsp)
   6:   movl   %eax, -0x4(%rsp)
   a:   retq
   b:   nopl   (%rax,%rax)
0000000000000010 <bar>:
  10:   movl   $0x5, %edi
  15:   jmp    0x1a <bar+0xa>
```

```
$ cat t.c
__attribute__((weak)) int foo() { return 5; }
int bar(void) { return foo(); }
$ clang -O2 -c t.c
$ llvm-objdump --no-show-raw-insn -d t.o
```

```
0000000000000000 <foo>:
   0:   movl   $0x5, %eax
   5:   retq
   6:   nopw   %cs:(%rax,%rax)
0000000000000010 <bar>:
  10:   jmp    0x15 <bar+0x5>
```

Gcc: Partial Inlining

- .part.<n> suffix
- Some part of control flow of a big function is inlined, but the rest of the function is not inlined.
- Clang also has a Partial Inlining pass, but disabled by default.

Partial Inlining: Before

```
static int xsk_rcv_check(struct xdp_sock *xs, struct xdp_buff *xdp, u32 len)
{
    if (!xsk_is_bound(xs))
        return -ENXIO;
    if (xs->dev != xdp->rxq->dev || xs->queue_id != xdp->rxq->queue_index)
        return -EINVAL;
    if (len > xsk_pool_get_rx_frame_size(xs->pool) && !xs->sg) {
        xs->rx_dropped++;
        return -ENOSPC;
    }
    sk_mark_napi_id_once_xdp(&xs->sk, xdp);
    return 0;
}
```

Partial Inlining: After

```
static int xsk_rcv_check(struct xdp_sock *xs, struct xdp_buff *xdp, u32 len)
{
    if (xs->dev != xdp->rxq->dev || xs->queue_id != xdp->rxq->queue_index)
        return -EINVAL;
    if (len > xsk_pool_get_rx_frame_size(xs->pool) && !xs->sg) {
        xs->rx_dropped++;
        return -ENOSPC;
    }
    sk_mark_napi_id_once_xdp(&xs->sk, xdp);
    return 0;
}
```

Caller:

```
    if (!xsk_is_bound(xs))
        return -ENXIO;
    ret = xsk_rcv_check(sx, xdp, len);}
```

...

Partial Inlining: Asm code

```
< for partial inlining, no information in dwarf >
ffffff840079c0 <xsk_rcv_check.part.0>:
; static int xsk_rcv_check(struct xdp_sock *xs, struct xdp_buff *xdp, u32 len)
ffffff840079c0: e8 4b 9b 10 fd    callq 0xfffffff81111510 <__fentry__>
ffffff840079c5: 41 56            pushq %r14
ffffff840079c7: 41 89 d6        movl  %edx, %r14d
ffffff840079ca: 41 55            pushq %r13
ffffff840079cc: 41 54            pushq %r12
ffffff840079ce: 55              pushq %rbp
ffffff840079cf: 53              pushq %rbx
ffffff840079d0: 48 89 fb        movq  %rdi, %rbx
ffffff840079d3: 48 83 ec 08     subq  $0x8, %rsp
;    if (xs->dev != xdp->rxq->dev || xs->queue_id != xdp->rxq->queue_index)
ffffff840079d7: 48 81 c7 08 03 00 00 addq  $0x308, %rdi    # imm = 0x308
```


Gcc: Constant Propagation

```
$ cat System.map | grep constprop | grep jhash
ffffffff81680ab0 t jhash.constprop.0
ffffffff82d31870 t jhash.constprop.0
ffffffff83429490 t jhash.constprop.0
ffffffff83722ad0 t jhash.constprop.0
...

include/linux/jhash.h
static inline u32 jhash(const void *key, u32 length, u32 initval)
{

// initval could be a known constant
// function signature for jhash.constprop.0 may change to
// jhash.constprop.0(const void *key, u32 length)
```

Gcc: Constant Propagation

Use dwarf to identify call site signature.

```
0x00f89b91:      DW_TAG_call_site
                  DW_AT_call_return_pc (0xffffffff811d3d6e)
                  DW_AT_call_origin   (0x00fa3b21 "jhash")
                  DW_AT_sibling (0x00f89baf)
0x00f89ba2:      DW_TAG_call_site_parameter
                  DW_AT_location   (DW_OP_reg5 RDI)
                  DW_AT_call_value (DW_OP_breg14 R14+0)
0x00f89ba8:      DW_TAG_call_site_parameter
                  DW_AT_location   (DW_OP_reg4 RSI)
                  DW_AT_call_value (DW_OP_breg13 R13+0)
0x00f89bae:      NULL
```

```
0x00f89baf:      DW_TAG_call_site
                  DW_AT_call_return_pc (0xffffffff811d4441)
                  DW_AT_call_origin   (0x00fa3b21 "jhash")
                  DW_AT_sibling (0x00f89bcd)
0x00f89bc0:      DW_TAG_call_site_parameter
                  DW_AT_location   (DW_OP_reg5 RDI)
                  DW_AT_call_value (DW_OP_breg3 RBX+8)
0x00f89bc6:      DW_TAG_call_site_parameter
                  DW_AT_location   (DW_OP_reg4 RSI)
                  DW_AT_call_value (DW_OP_breg13 R13+0)
0x00f89bcc:      NULL
```

Gcc: Interprocedural Scalar Replacement of Aggregates

- .isra.<n>
- Perform interprocedural scalar replacement of aggregates, removal of unused parameters and replacement of parameters passed by reference by parameters passed by value.
- **Function signature may change**

Gcc: Interprocedural Scalar Replacement of Aggregates

```
static struct mem_cgroup *bpf_map_get_memcg(const struct bpf_map *map)
{
    if (map->objcg)
        return get_mem_cgroup_from_objcg(map->objcg);
    return root_mem_cgroup;
}
```

TO

```
static struct mem_cgroup *bpf_map_get_memcg(struct obj_cgroup *objcg)
{
    if (objcg)
        return get_mem_cgroup_from_objcg(objcg);
    return root_mem_cgroup;
}
```

Gcc: Interprocedural Scalar Replacement of Aggregates

The number of arguments may not change. In such cases need to inspect asm code.

```
ffffff815882c0 <bpf_map_get_memcg.isra.0>:
; static struct mem_cggroup *bpf_map_get_memcg(const struct bpf_map *map)
ffffff815882c0: e8 4b 92 b8 ff    callq  0xffffffff81111510 <__fentry__>
;   if (map->objcg)
ffffff815882c5: 48 85 ff          testq  %rdi,%rdi
; static struct mem_cggroup *bpf_map_get_memcg(const struct bpf_map *map)
ffffff815882c8: 41 57            pushq  %r15
ffffff815882ca: 41 56            pushq  %r14
ffffff815882cc: 41 55            pushq  %r13
ffffff815882ce: 41 54            pushq  %r12
ffffff815882d0: 55              pushq  %rbp
ffffff815882d1: 53              pushq  %rbx
;   if (map->objcg)
ffffff815882d2: 75 38           jne   0xffffffff8158830c <bpf_map_get_memcg.isra.0+0x4c>
;   return root_mem_cggroup;
ffffff815882d4: 48 c7 c0 58 10 7e 86  movq  $-0x7981efa8, %rax  # imm = 0x867E1058
```

Automating special case function handling in pahole, libbpf and kernel

- We will discuss
 - How pahole handles optimizations today
 - How it might do better in the future
- And
 - How pahole handles multiple functions with same name, different function signature
 - And how it might do better in the future
- Aim is always to ensure accuracy (even if this sacrifices tracing some functions)
- But we will see that having more info in BTF may allow us to trace more *without* sacrificing accuracy

How pahole handles optimizations today

Focus is on making tracing experience consistent with expectations based upon code inspection.

This is done as it gives us a consistent view regardless of -O level.

...and it makes things safer for the verifier by not providing BTF for problem functions.

Less interested in hot-cold splits; latter are not the real function entry point.

Most interesting case is “.isra.” suffix since this often preserves parameters and is function entry point.

gcc, optimizations and DWARF

Not well documented but in `gcc/dwarf2out.cc` we see in `gen_subprogram_die()`:

```
/* This function gets called multiple times for different stages of  
the debug process. For example, for func() in this code:
```

```
void func() { ... }
```

...we get called 4 times. Twice in early debug and twice in late debug:

Once for `func()` itself. As in (2), this is the specification, but this time we will re-use the cached DIE, and just annotate it with the location information that should now be available.

So for an `.isra` function, we can correlate the abstract origin description with the original function name; and note which arguments actually refer to registers to determine register/argument relationship.

How pahole handles optimizations today

Late DWARF generation in gcc gives us location information about function parameters after optimization (see previous slide)

We use this to verify that function parameters are registers we expect, and do not encode `.isra/.constprop` functions with unexpected register or constant use

So “`func.isra.0`” gets a BTF representation if and only if its parameter register use matches the function signature

The BTF representation generated (`BTF_KIND_FUNC`) is named “`func`” – the suffix is omitted; gives us a more stable representation.

Future work - tracing optimized functions

Problem today is we only have names to connect BTF and kallsyms, and name may refer to multiple instances in kallsyms (foo, foo.isra.0, foo.constprop.0 etc)

We can add support to tie the BTF + kallsyms reference; this would allow programs that specify

```
SEC("fentry/func")
```

...to just work, even when "func" was optimized to be "foo.isra.0".

We know the signature matches expectations – even if it is optimized - since pahole checks during BTF generation

Just need to use address or an alias BTF tag to link function BTF + kallsyms representations

Future work - tracing inlined functions

Inlined functions are tricky

Sheer numbers of sites make BTF representation expensive

...and since when inlined, register usage is hard to correlate with function call expectations

We need a better answer for this; Linux is so heavily inlined.

Need a way to represent function-associated addresses in BTF

Future work – inlined functions parameters

Measurements show that DWARF often preserves locations of inlined function parameters and that these locations are often “simple”:

- A register [+ constant]
- A constant

E.g. for BPF selftests vmlinux* ~350K inline “callsites” are recorded, of these 67% have information about all parameters and these parameters are in “simple” locations **.

Representing this information in BTF would allow to create trampolines setting parameters information for “krobe_maybe_inline” programs.

* compiled using LLVM20.0.0.git for x86

** measured using <https://github.com/eddyz87/inline-address-printer>

Future work – tracing optimized functions

LTO considerations?

Static function promotion with no signature change in thinLTO (.llvm.1234) should be doable

As noted already fixed for BTF/kallsyms comparison

<https://lore.kernel.org/r/20240326041458.1198161-1-yonghong.song@linux.dev>

fentry/fexit support needed

What about full LTO and func.1234?

Can we trust function signature is unchanged? If not how can we check? Late DWARF in LLVM?

How pahole handles inconsistent functions today

Approx 100 static functions share the same name but have inconsistent function prototypes

A problem for BPF and the verifier, since we rely on BTF to tell us what the function parameters are

pahole handles this by not adding BTF representation for such inconsistent functions

Future work – inconsistent functions

Having additional address information in BTF would allow us to handle inconsistent functions. Consider

```
static int func(int arg);          // in CU 1
static int func(struct sk_buff *arg); // in CU 2
```

With address info, we could match up kallsyms+BTF to trace the right function based upon function signature; so we could trace CU 2 version with

```
SEC("fentry/func")
int BPF_PROG(tracefunc, struct sk_buff *arg)
{
...
}
```

Future work - common needs

Handling both optimized + inconsistent cases would be good.

At present, BTF does not provide representations to support this.

Having address information in BTF would help resolve BTF->kallsyms relationships for both cases.

Addresses could also potentially mark inline sites, and allow us to associate function signatures with addresses.

Could we also provide information about

- inlined function parameters

- Case where parameter is not in the calling-convention-expected register?

Future work - encoding function addresses in BTF

Size considerations; 60,000 kernel functions

Avoid bloating BTF as much as possible

Don't want to leak absolute addresses to the world via `/sys/kernel/btf`

Use base-relative addresses

Preserve existing semantics where possible; `BTF_KIND_FUNC` -> `BTF_KIND_FUNC_PROTO` means the function has a stable function prototype at function entry.

Future work - encoding function addresses in BTF

Proposal: extend `BTF_KIND_DATASEC`

Currently it specifies

- an ELF section name
- A set of variables in the section (`btf_var_secinfo`)

Proposal

If `kind_flag` is set for `BTF_KIND_DATASEC`, it contains a set of struct `btf_func_secinfo` describing the functions

Future work - encoding function addresses in BTF

```
struct btf_func_secinfo {  
    __u32  type;      // BTF_KIND_FUNC  
    __u32  offset;    // relative offset address  
    __u32  location_type; // BTF_KIND_FUNC_PROTO or BTF_KIND_LOC  
};
```

BTF_KIND_LOC is a struct btf_type where the type points at the associated function prototype; it is followed a set of struct btf_loc {}, one for each function parameter. Each of these represents a string description of the location associated with the parameter, 0 if no location info is provided.

How will I tell a function is partially/fully inlined?

Answer: find a `BTF_KIND_FUNC` with the function name and a 0 type specifying no prototype. What that says is "I cannot associate a stable prototype with this function, you will need to consult site-specific data in `BTF_KIND_DATASEC` to trace this".

Benefits:

- it is easy to find functions that are safe to trace and satisfy calling conventions with respect to parameters;
- it is easy to spot inline functions; they have `BTF_KIND_FUNC` with a 0 type.
- tracing works as it does today; only functions with a `FUNC -> FUNC_PROTO` are assumed to be valid for tracing such that calling conventions are always observed

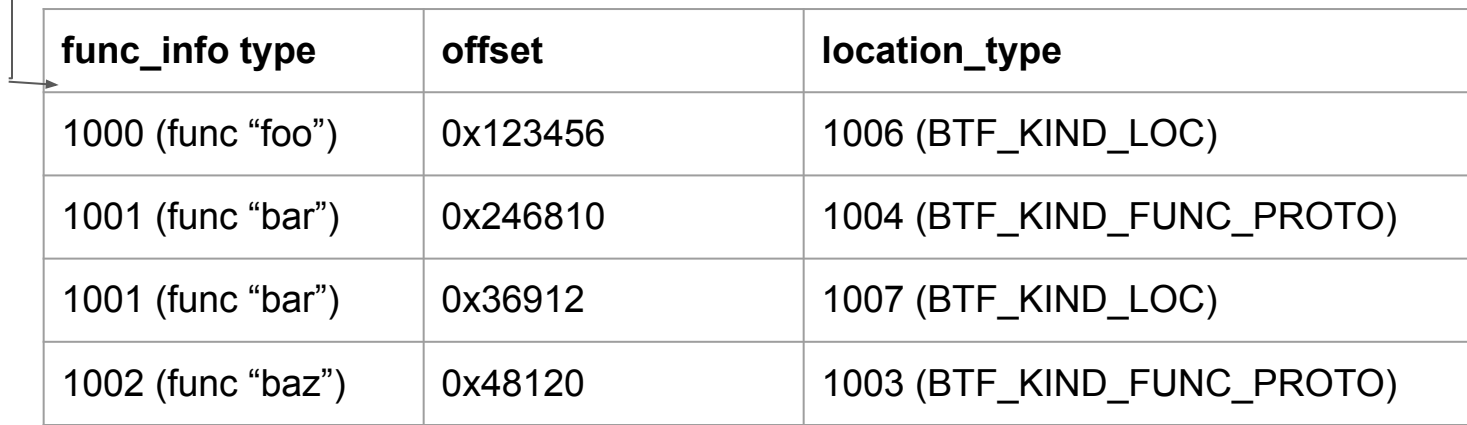
How will I find function info?

1. Lookup BTF_KIND_FUNC with name; if it has a 0 type we know that it is inlined or inconsistent wrt calling conventions
2. we look in BTF_KIND_DATASEC with kind_flag 1 (indicating function info) for a btf_func_secinfo specifying that BTF_KIND_FUNC. From there we can get the addresses of sites/location info mapping parameters to registers etc

ID	Name	Kind	Type ref
1000	foo	BTF_KIND_FUNC	0
1001	bar	BTF_KIND_FUNC	0
1002	baz	BTF_KIND_FUNC	1003
1003	-	BTF_KIND_FUNC_PROTO	...
1004	-	BTF_KIND_FUNC_PROTO	...

How will I tell a function is partially inlined/inconsistent?

ID	Name	Kind	Type ref
1000	foo	BTF_KIND_FUNC	0
1001	bar	BTF_KIND_FUNC	0
...
1005		BTF_KIND_DATASEC	-



An arrow points from the BTF_KIND_DATASEC entry (ID 1005) in the first table to the first row of the second table.

func_info type	offset	location_type
1000 (func "foo")	0x123456	1006 (BTF_KIND_LOC)
1001 (func "bar")	0x246810	1004 (BTF_KIND_FUNC_PROTO)
1001 (func "bar")	0x36912	1007 (BTF_KIND_LOC)
1002 (func "baz")	0x48120	1003 (BTF_KIND_FUNC_PROTO)

Where do we store this information?

- Adding BTF_KIND_DATASEC information for each function in the kernel means adding $60,000 * 12$ bytes -> 703Kb.
- Adding BTF_KIND_LOC info for each inconsistent function (~700) means roughly 32 bytes/loc (assumes ~5 parameters need location info) -> 21Kb
- Adding BTF_KIND_DATASEC information for each inline site with accessible parameters means $235,000 * 12$ bytes =2.8Mb
- Adding BTF_KIND_LOC info for each of these with roughly 32 bytes/loc -> 7Mb
- Total is bigger than total existing vmlinux BTF today, but would be so useful
- For inlines have option CONFIG_DEBUG_INFO_BTF_EXTRA
- Deliver it via a separate module
- Total vmlinux + extra info would still be <20Mb versus 400Mb of DWARF

Future work – optimizations + function addresses in BTF

For optimized functions, we can also match “func” BTF representation with kallsyms

If kallsyms lookup of exact name “func” fails, use BTF address to match relevant kallsyms prefix-matched entry

On BTF load we can sort BTF addresses to provide fast lookup

With this in place

SEC(“fentry/foo”)

...would work, even if it is foo.isra.0

Would help with llvm foo.1234 suffixes too

Future work – tracing inline sites via kprobes

For inlined functions we can use address info and location information to provide multiple attach to sites with mapping of parameters->locations

Similar to what is done for USDT today.

Could automate attach with a custom ELF section name, i.e.

`SEC("kprobe_maybe_inline/foo")`

Even with no location data, we can use kprobe tracing to see when inline functions are “called”.

Future work – inconsistent functions + addresses in BTF

For inconsistent functions we could now use the BTF representation to match kallsyms entries for inconsistent functions:

```
SEC("fentry/func")
```

```
int BPF_PROG(tracefunc, struct sk_buff *arg)
```

```
{
```

```
...
```

Match via name+BTF function signature, then use BTF_KIND_FUNC id to get address in BTF

With that, we can trace the right instance(s)

We'd need to think about multi-attach semantics (do by default? Or if requested)

Summary

Problem	Description	gcc	LLVM?	Solution
.isra	Interprocedural scalar replacement...	Yes	Yes	late DWARF allows us to see inconsistent info, use locations
.constprop	Constant propagation	Yes	Yes	Providing location info will enable per-site handling of constant args
Inlining	Functions “disappear”	Yes	Yes	Site addresses + prototypes + location info
Inconsistent funcs	Same name, multi proto	Yes	Yes	Per-site clarity on which proto used
Thin LTO .llvm.num suffixes	Promotion does not change function signature	No	Yes	Address maps name->func

For more info

BTF spec <https://www.kernel.org/doc/Documentation/bpf/btf.rst>

pahole support for optimizations

<https://lore.kernel.org/all/1675088985-20300-1-git-send-email-alan.maguire@oracle.com/>

More CO-RE? Functions, optimizations and ensuring trace accuracy

<https://lpc.events/event/16/contributions/1371/attachments/1020/2090/More-CO-RE-LPC2022.pdf>

<https://github.com/eddyz87/inline-address-printer>