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

Alan Maguire
Linux Kernel Networking, Oracle
alan.maguire@oracle.com
blogs.oracle.com/linuxkernel
September 2022

Making use of the BPF Type Format

We are using BTF more and more in kernel observability and debugging.

As such, we need to ensure we have as complete a picture as possible in our BTF representation – we will focus on kernel functions here.

To be clear - this presentation is fairly far in the weeds – the vast majority of kernel functions are represented correctly in BTF (depending on what optimization level you build your kernel at, you may see few of these issues – if any!)

However it is critical that tracing provides a complete picture that preserves accuracy *where possible*.

Specific Motivation

Most distros enable some level of optimization.

Long-standing issues in tracing “isra breaks kprobes”

- https://github.com/iovisor/bcc/issues/1754

Surprising how often we see issues with “missing” functions, or unexpected argument values.

Compile Once – Run Everywhere has made running BPF programs on different kernels *much* easier by insulating us from type offset changes.

Maybe some of the same mechanisms might help here?

Let’s start by seeing how complete our BTF representation is.

How completely does BTF cover kernel functions?

Because /proc/kallsyms lumps variables and functions together as text (T/t) objects, we can use objdump to get a better count of core kernel functions:

```
$ objdump -t vmlinux | awk '/ F / { print $0}'|wc -l
  55774
```

And counting BTF_KIND_FUNCs for kernel vmlinux:

```
$ bpftool btf dump file vmlinux | awk '/ FUNC /
{ print $3 }' | wc -l
  48790
```

So we are missing BTF info for 6984 functions – approximately 1 in 8 kernel functions; what are these missing functions?

Remaining count: 55774 – 48790 = 6984

Duplicate function definitions:

Many static functions have multiple definitions in the kernel

```
$ objdump –t vmlinux | awk '/ F / { print $6}'|sort|uniq|
wc –l

55055
```

…which brings our discrepancy down to 6265 functions.

These have identical function signatures, so aren't a BTF issue, but do present a problem – which do I attach to?

Might make sense to have an "attach-all" option, as we might miss events if we attach to just one.

Remaining count: 55055 – 48790 = 6265

Unlikely-to-run .cold functions

The largest subset of these – over 50% - are “.cold”-suffixed functions

```
$ objdump –t vmlinux | awk '/ F / { print $6}'|sort |
uniq|grep ".cold" |wc –l

3342
```

We notice in kallsyms, a function is always paired with a .cold suffixed function of the same name:

ffffffff8a477d50 t a4_probe

ffffffff8a75b2d8 t a4_probe.cold

What are these?

Unlikely-to-run .cold functions

https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html

The cold attribute on functions is used to inform the compiler that the function is unlikely to be executed. The function is optimized for size rather than speed and on many targets it is placed into a special subsection of the text section so all cold functions appear close together, improving code locality of non-cold parts of program. The paths leading to calls of cold functions within code are marked as unlikely by the branch prediction mechanism. It is thus useful to mark functions used to handle unlikely conditions, such as perror, as cold to improve optimization of hot functions that do call marked functions in rare occasions.

Unlikely-to-run .cold functions

We don’t have DWARF information about .cold function signatures.

Looking at examples – e.g. a4_probe[.cold] – the .cold function is used to handle unlikely error paths such as the highlighted one below:

```
a4 = devm_kzalloc(&hdev->dev, sizeof(*a4), GFP_KERNEL);
if (a4 == NULL) {
        hid_err(hdev, "can't alloc device descriptor\n");
        return -ENOMEM;
}
```

Remaining count: 6265 – 3342 = 2923

Partially-inlined .part.N functions

The next largest subset - are ".part.N" functions. These overlap with .cold functions, so let's eliminate them from our count:

```
$ objdump –t vmlinux | awk '/ F / { print $6}'|sort |
uniq|egrep –v .cold|grep ".part."|wc –l

954
```

Again we notice in kallsyms, a function is often paired with a .part variant:

```
ffffffffc108ccf0 t account_huge_nx_page.part.0  [kvm]

ffffffffc1093bb0 t account_huge_nx_page  [kvm]
```

Note the address of the .part.0 is prior to the associated function. What are these?

Partially-inlined .part.N functions

```
/* The purpose of this pass is to split function bodies to improve inlining. I.e. for function:

     func (…) {
          if (cheap_test)
               something_small
          else
               something_big
     }

Produce:

     func.part (…) {
          something_big
     }
     func (…) {
          if (cheap_test)
               something_small
          else
               func.part (...);
     }
```

https://github.com/gcc-mirror/gcc/blob/master/gcc/ipa-split.cc

Partially-inlined .part.N functions

.part functions are generated from bigger funtions which cannot be inlined in their totality. The aim is to inline the inline-able part at call sites where possible, and the non-inlined .part then gets called from these.

We can see in many cases (390/1385), the parent function is *not* inlined, and we have both parent and .part.0, but sometimes we just have the .part.0 too:

ffffffff89e6b590 t account_event_cpu.part.0

In the above case, the actual function *site* was inlined.

So tracing this != tracing account_event_cpu()

Remaining count: 2923 – 954 = 1969

Scalar replacement, parameter-modifiying .isra.N functions

```
$ objdump –t vmlinux | awk '/ F / { print $6}'|egrep –
v .part |egrep –v .cold |sort |uniq|grep ".isra" |wc –l

833
```

-fipa-sra

Perform interprocedural scalar replacement of aggregates, removal of unused parameters and replacement of parameters passed by reference by parameters passed by value.

https://github.com/iovisor/bcc/issues/1754 (“ISRA breaks kprobes”)

These represent a case where optimization may diverge from the function signature.

Remaining count: 1969 – 833 = 1136

Scalar replacement, parameter-modifiying .isra.N functions

```
#include <stdio.h>

struct foo {
        int val1;
        int val2;
};

static noinline void doit(struct foo *foo, int val1, int val2) {
        printf("got val1 %d val2 %d\n", val1, val2);
}

int main(int argc, char *argv[]) {
     struct foo foo;
     int i = 1;
     foo.val1 = argc; foo.val2 = I;
     doit(&foo, foo.val1, foo.val2); // foo optimized out, val1 first arg, val2 constant
     return 0;
```

Scalar replacement, parameter-modifiying .isra.N functions

We see that the function description in DWARF reflects function signature in source, not optimizations:

```
<1><3d7>: Abbrev Number: 25 (DW_TAG_subprogram)
   <3d8>   DW_AT_name        : (indirect string, offset: 0x77): doit
 <2><3e4>: Abbrev Number: 26 (DW_TAG_formal_parameter)
  <3e5>   DW_AT_name        : foo
  <3ec>   DW_AT_type        : <0x409>
<2><3f0>: Abbrev Number: 27 (DW_TAG_formal_parameter)
  <3f1>   DW_AT_name        : (indirect string, offset: 0x7c): val1
  <3f8>   DW_AT_type        : <0x5f>
<2><3fc>: Abbrev Number: 27 (DW_TAG_formal_parameter
  <3fd>   DW_AT_name        : (indirect string, offset: 0x2ad): val2
    <404>   DW_AT_type        : <0x5f>
```

Scalar replacement, parameter-modifiying .isra.N functions

There is still hope! Later subprogram info refers to the same function tag:

```
<1><40f>: Abbrev Number: 28 (DW_TAG_subprogram)
   <410>   DW_AT_abstract_origin: <0x3d7>            <-- this is the doit function
 <2><42a>: Abbrev Number: 29 (DW_TAG_formal_parameter)   <-- this is val1, and location value is
                                                 000000e2 0000000000401150 000000000040115c (DW_OP_reg5 (rdi))
    <42b>   DW_AT_abstract_origin: <0x3f0>
    <42f>   DW_AT_location    : 0xe2 (location list)
 <2><437>: Abbrev Number: 29 (DW_TAG_formal_parameter) <-- this is foo, and looking at location entry
    00000130 0000000000401150 0000000000401163 (DW_OP_GNU_parameter_ref: <0x3e4>; DW_OP_stack_value)
    <438>   DW_AT_abstract_origin: <0x3e4>
    <43c>   DW_AT_location    : 0x130 (location list)
 <2><444>: Abbrev Number: 30 (DW_TAG_formal_parameter) <-- this is val2; we see it is const so not passed into doit
    <445>   DW_AT_abstract_origin: <0x3fc>
    <449>   DW_AT_const_value : 1
```

Scalar replacement, parameter-modifiying .isra.N functions

This isn't very well documented but in gcc/dwarf2out.cc we see comment 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:
…
     4. 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.

Constant propagation .constprop.N functions

```
$ objdump –t vmlinux | awk '/ F / { print $6}'|egrep –
v .part |egrep –v .cold |egrep –v .isra | sort |uniq|grep
".constprop" |wc –l

284
```

These represent (possibly multiple copies of the same) function with (conflicting) constant parameters.

These again represent a case where optimization may not reflect the function signature.

Remaining count: 1136 - 284 = 852

Remaining functions

Counting cases discussed so far:

```
$ objdump -t vmlinux | awk '/ F / { print $6}'|grep -E
"(\.part|\.cold|\.constprop|\.isra)"|sort |uniq|wc -l
```

5341

Remaining count: 6265 – 5341 = 924

Of these 924, 739 are declaration-only static call trampolines used to mitigate SPECTREv2, skipped by BTF generation:

```
$ grep __SCT_ missing_btf_funcs |wc -l
```

739

Remaining count: 924 – 739 = 185

Remaining functions

Functions prefixed by xen_hypercall_ are not in DWARF, and these amount to

```
$ grep xen_hypercall missing_btf_funcs |wc -l
49
```

Remaining count: 185 – 49 = 136

Many of the remaining functions – 87 to be exact - are assembly only, so don't have DWARF (arch_rethook_trampoline, asm_load_gs_index, …); 10 more functions are declaration-only (so are skipped for BTF generation); and the remaining 39 are __ia32 __i64 syscalls that have no DWARF representation.

Remaining count: 136 – 87 – 10 - 39 = 0

So we have accounted for all kernel function symbols!

What to do – easier cases

.cold (3342) and .part (954) functions have no DWARF info to convert to BTF (unless there is just one .part.0 function replacing the original).

Declaration-only (749) functions are explicitly skipped for BTF generation.

Assembly functions (87) have no DWARF → no BTF conversion is possible.

Duplicate static function names (700) – perhaps support multiple attach option to cover attaching to all of them? Or kallmodsyms-like solution?

What to do – harder cases

.isra (833), .part (954) and .constprop (284) can change function args.

.constprop, .part are hard to to handle, given that

- multiple representations for different constants may be used for the same base function; and
- multiple sets of these static functions may exist too.

Possible for .constprop (where *one* .constprop replaces the original).

Avoid handling .part, since tracing might miss the inlined prologue.

It seems that the most feasible path is for .isra functions.

If so, we have some decisions to make:

Should BTF reflect optimizations or original function signature?

And if not via BTF FUNC_PROTO, how would we represent optimization changes?

CO-RE for functions to the rescue?

CO-RE principles would argue for a more stable representation, not dependent on compiler optimization.

Ideally we want BPF users to be able to write portable programs based on function signature and hide the details where possible, similar to CO-RE for types.

Proposal 1: BTF retains non-optimized function signature from DWARF, while providing additional annotations to allow tracing programs to get accurate argument data where possible.

Proposal 2: libbpf uses these annotations to hide optimization issues from consumers where possible using CO-RE mechanisms

Function argument annotations via declaration tags?

- BTF provides support for declaration tags that can reference a specific argument in a function signature
- Having conventions around tag names containing information such as “arg2 is optimized out”, “arg3 is const 1” seems possible..
- Perhaps we could use the kind flag in the BTF_KIND_FUNC to signal the presence of function argument declaration tags?
- If the flag is set for a tracing function, libbpf knows it needs to do CO-RE relocations based upon declaration tags.
- How to handle optimized-out values? Zero (like exception handling)?

Proof-of-concept

A version of pahole that

- Finds DWARF subroutine parameters that use DW_AT_abstract_origin to point at original function parameter; then
- Retrieve basic location information; if not a register for args < max # register args, we flag it as “optimized out”
- When generating BTF, we can use that info
- Current version just skips such args in BTF, but that’s not ideal

  https://github.com/alan-maguire/dwarves/tree/btf-optimization

Proof-of-concept example

static ssize_t __init xwrite(struct file *file, const unsigned char *p,
size_t count, loff_t *pos);

Becomes xwrite.constprop.0

Callers pass “pos” as pointer to static variable:

```
static __initdata loff_t wfile_pos;
```

Late debuginfo for xwrite shows:

<b324e> DW_AT_location : 10 byte block: 3 40 72 48 83 ff ff ff ff 9f (DW_OP_addr: ffffffff83487240; DW_OP_stack_value)

pahole PoC spots the fact that this ^^ isn’t a register

Non-optimized functions and parameter checking

This version of pahole also identified some non-suffixed functions where stack was used for args instead of registers

```
ZSTDLIB_API ZSTD_DStream*
ZSTD_createDStream_advanced(ZSTD_customMem customMem);
```

customMem is a larger struct, passed via the stack.

Recent support for struct support for trampoline-based progs handles small (register-passed) struct case; maybe we could use these mechanisms to handle the larger struct case too?

Using tag information from BTF to fixup function args

Can we emit relocations for tagged arguments, to fix them up to look right?

Could we hack PT_REGS_PARMx_CORE() to point at the right things, based upon parameter tagging?

Assumes we know what we are attaching to, so would need to be attach-time..

Conclusion

Handling optimizations is hard!

- copy_query_item.isra.0.part.0.constprop.0

Ensure tracing semantics make sense; if not, don’t represent at all.

Separating function signature – more stable – from optimizations seems most CO-RE-like behavior.

Late DWARF debuginfo provides mechanism to spot optimizations.

BTF declaration tags seem like a good mechanism to allow us to annotate the function prototype.

References

Good summary of problem “isra breaks kprobes”

https://github.com/iovisor/bcc/issues/1754

Proof-of-concept for spotting optimized parameters

https://github.com/alan-maguire/dwarves/tree/btf-optimization


ORACLE®