BPF tracing: exploring additional debugging capabilities

Alan Maguire
Linux Kernel Networking, Oracle
alan.maguire@oracle.com
blogs.oracle.com/linuxkernel
September 2021

Idea: gdb can provide inspiration for BPF features

- gdb has a deep history with making debugging software easier
- But many problems are not amenable to the linear form of debugging that gdb supports
- This will probably get worse as the world gets more parallel
- BPF tracing is needed for such problems
- ...but it is not as feature-rich (yet!)
- Important to keep in mind tradeoffs

Program Agenda

1. Deep data structure display
2. Macros and BTF
3. Inline functions
4. Local variables

ORACLE

1. Deep data structure display

- Debuggers such as gdb have excellent support for examining data.

```
(gdb) print test_cases
$1 = {root = {rb_node = 0xffff8800d1d16800}, size =
1,
  keycompare = 0xffffffffa0bdbe10
<ktf_map_name_compare>}
```

1. Deep data structure display and BPF/BTF

- libbpf provides btf_dump*() functions to dump BPF Type Format-based type descriptions
- Used for header generation simplifying BPF program writing

```
# bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

- New interface recently added – btf_dump__dump_type_data() - allows us to
  - dump a representation of provided data
  - …using BTF information associated with its type

1. Deep data structure display and BPF/BTF

- Example output:

```
(struct sk_buff){
     (union){
             (struct){
                     .next = (struct sk_buff *)0xffffffffffffffff,
                     .prev = (struct sk_buff *)0xffffffffffffffff,
             (union){
                     .dev = (struct net_device *)0xffffffffffffffff,
                     .dev_scratch = (long unsigned int)18446744073709551615,
             },
     },
 ...
```

1. Deep data structure display – future work

- Tracing tool integration (bpftrace, perf, BPF-based DTrace?)
  - See ksnoop for an example of tracing with deep data display

    https://github.com/iovisor/bcc/tree/master/libbpf-tools/
- Syzkaller/ksnoop integration? Capture args for syzkaller-induced failures
  - Once we have reproducer for failure fn1->fn2->fn3->fn4, run “ksnoop -s fn1 fn2 fn3 fn4”
- Diff-style output? Useful for tracking changes in a data structure over time
  - Pass in “template” data/size as an option to btf_dump__dump_type_data()
  - As the data structure is traversed, fields are compared, and where data differs from template data:

```
<    .foo = (int)1
--
>    .foo = (int)2
```

2. Macros and BTF

- When writing BPF programs, we can generate a header (such as vmlinux.h) to capture types from kernel
- Split BTF allows the same for modules now
- header generation + CO-RE – along with bpftool skeletons - means that BPF program development is much easier now - less wrestling with kernel headers
- But there is one piece missing…
- #defines!

2. Lessons from GDB - macros

- gcc supports a -ggdb3 option to add a .debug_macro section to DWARF (v2+). For example:

```
#ifdef CONFIG_FOO
#define F1       0x1
#else
#define F1      0x2
#endif
#define F2(a)    (a > 0)
```

  ..becomes (via “objdump -Wm”, showing the .debug_macro section)

```
DW_MACRO_GNU_define_indirect - lineno : 3 macro : F1 0x2
DW_MACRO_GNU_define_indirect - lineno : 4 macro : F2(a) (a > 0)
```

- Note actual, not possible definitions are used. gdb can reference macros for debugging

```
(gdb) p task_is_stopped_or_traced(init_task)
```

2. Cost of macro encoding

- Cost in BTF size: vmlinux built with -ggdb3 contains 1,666,106 definitions.
  - Many of these are duplicated; de-duplication (prior to BTF conversion) helps here 1,666,106 → 81,095
    - BTF encoding could simply be

      ``struct btf_type`` encoding requirement:

      * ``name_off``: any valid offset (pointing at a `DW_MACRO_GNU_define_indirect` string for name)

      * ``info.kind_flag``: 0

      * ``info.kind``: BTF_KIND_MACRO

      * ``info.vlen``: 0

      * ``val_off``: offset for value string (added to anon union of size/type).
    - Eventual vmlinux BTF size using this scheme is 9.6MB with deduplicated macros versus ~6.3MB without.

2. Macro encoding mechanics

- We need some rules in processing macros
  - De-duplication of identical definitions helps; what about non identical?
  - Heuristic used: if the same macro name has multiple different values, do not encode it in BTF
  - Similarly, we avoid encoding macros which have associated “#undef”s, as this is a strong signal of too-clever-by-half
  - Exclude CONFIG_ variables, a few definitions “inline”, “noinline”, “container_of”
  - We end up with ~68,000 definitions using these rules.
  - What about split BTF? Apply same rules as above but including base BTF.
- vmlinux.h format consists of
  - #ifndef __VMLINUX_H__ … #endif -guarded section for types
  - #ifndef __VMLINUX_MACROS_H__ … #endif section for macros.

2. BPF program use of vmlinux.h (**bpf_iter_task_vma.c)**

```
#include "vmlinux.h"
#ifndef __VMLINUX_MACROS_H__
/* Copied from mm.h */
#define VM_READ         0x00000001
#define VM_WRITE        0x00000002
#define VM_EXEC         0x00000004
#define VM_MAYSHARE     0x00000080
/* Copied from kdev_t.h */
#define MINORBITS       20
#define MINORMASK       ((1U << MINORBITS) - 1)
#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
#endif
```

2. Problem – What about CO-RE?

- CO-RE: how do we handle #define changes for different kernel versions?
- A potential solution for numeric values. Consider

  #define FLAG_VAL   0x1

  - In our BPF program, we specify a global definition such as

    BPF_CORE_DEFINE(FLAG_VAL, int);

    ...which expands to

    static int BPF_CORE_VAR_FLAG_VAL = FLAG_VAL;
  - This is a placeholder value which is initialized to FLAG_1's current value. On BPF program load libbpf fills in the value for the current kernel from the BTF definition in the .data section of the ELF file, similar to how type ids are filled in.
  - So definitions become variables initialized to the appropriate numeric value for use in programs.
  - libbpf would have to take string definition and convert it to appropriate type.

3. Inline functions

- The Linux kernel makes extensive use of inline functions, so some functions aren’t easily traceable due to inlining
- However, we can trace most instructions using kprobe+offset
- …So if we know the addresses of the inline sites, we can potentially attach kprobes to fire for all instances of an inlined function
- What we want; trace all sites of inline_function:

  SEC(“kprobe/inline_function”)

  …and perhaps to trace a specific inline site

  SEC(“kprobe/inline_function@containing_function”)

3. Inline functions

- gdb uses DWARF information to identify inlines + sites
- So can we augment BTF with inline function info?
- We could encode inline sites via BTF id of calling function + 32-bit offset/len
- So a BTF_KIND_FUNC could use a kind_flag to specify INLINE, in which case
  - info.vlen specifies number of sites following BTF_KIND_FUNC
  - Each btf_inline_site is

```
struct btf_inline_site {
    __u32 func_btf_id;
    __u32 unused;
    __u32 func_offset;
    __u32 func_len;
};
```

3. Inline functions

- So with this encoding, the algorithm for a SEC(“kprobe/func”) becomes:
  - Look up the function name in kallsyms
  - If present, it’s a “real function”
  - If not present, look it up as BTF_KIND_FUNC, and if it is marked inline
    - foreach btf_inline_site
      - Attach a kprobe at caller’s kallsyms address + offset

3. Cost of inlines for BTF

- DWARF information on inline sites seems to be incomplete (*)
- For vmlinux
  - There are 35949 inline functions
  - For these, there are 86027 inline sites with lo-hi Program Counter values
  - A further 215773 inlines have no associated lo-hi PC values
- From a BTF perspective, encoding costs of available info are
  - 16 bytes per BTF_KIND_FUNC (16 x 35949=575184)
  - 16 bytes per inline site (16 x 86027= 1376432 available, 16x215773=3452368 possible)
  - = 1.86Mb for available, 3.84Mb for possible

  (*) this may be a result of cluelessness in using libdw/libdwfl on my part

3. Problems

- BTF inline sites will not expose arguments via “struct pt_regs” in the same way a function entry point does
- As such, it may not be necessary to follow a BTF_KIND_FUNC inline definition with a BTF_KIND_FUNC_PROTO description of args etc
- For split BTF, deduplication may have to preserve a FUNC definition in base and split BTF since kernel modules will have their own set of inline sites.

4. Local variables

- BTF contains information about kernel data
  - Global variables, per-CPU variables, function arguments/return values
- The only missing piece here is information about local variables.
- Local variable info would be useful for trace-based stepping through functions, or simply attaching to a specific offset into a function (inline site)
  - At a kprobe other than entry, what do registers contain?
  - Helper such as bpf_whatis(ip, &reg_btf_ids) could retrieve type ids for struct pt_regs in such cases
  - Program can then look through the reg list for the “struct sk_buff”.
  - True instruction-level tracing

4. Problems

- DWARF encodes variables using DW_TAG_variable
- Locations of variables are specified using DW_AT_location, which can point to a list of locations where the variable lives over the lifetime of the function (stackframe-base offsets, registers)
- For vmlinux there are 2,005,353 DW_TAG_variable instances.
- Even without location information, representing these would significantly increase the size of vmlinux BTF.
- Just a "struct btf_type" for each would cost >30Mb
- Current vmlinux BTF size is ~6-7Mb.

4. Alternatives to encoding all local variables?

- Perhaps we can utilize BTF information about functions to do dynamic type inference?
  - From BTF, we know BTF ids associated with argument registers and return value
  - From here, we can build a graph of how register values evolve
  - Perhaps we can re-use some of the verifier magic to scan kernel functions in this way?
  - Some missing pieces: some local variables will not connect to args, return value
  - Maybe we just need to encode those variables to reduce size costs

Summary

- BPF has made huge strides in observability/usability, many powered by BTF
- BPF/BTF can grow further capabilities, and gdb is a good source of ideas
- But cost-benefit must always respect key benefits of BTF which are
  - Size
  - Simplicity
  - Always present (for CONFIG_DEBUG_INFO_BTF kernels at least)
- For more expensive representations, perhaps hybrid dynamic/static methods for creating descriptions are worth exploring

References

- For more on dumping data using BTF, see my eBPF summit talk:
  - Data-centric tracing using BTF: https://www.youtube.com/watch?v=PRiJYuDMtEQ
- ksnoop, an example of data-centric tracing
  - https://github.com/iovisor/bcc/tree/master/libbpf-tools/
- Discussion of gdb and macros. The -ggdb switch seems to be very lightly documented on the web; I found the first reference here: https://blogs.oracle.com/linux/post/8-gdb-tricks-you-should-know
- TBD: RFC patchset for bpf-next + RFC dwarves patch for macro support.