NAME

pahole − Shows, manipulates data structure layout and pretty prints raw data.

SYNOPSIS

pahole [*options*] *files*

DESCRIPTION

pahole shows data structure layouts encoded in debugging information formats, DWARF, CTF and BTF being supported.

This is useful for, among other things: optimizing important data structures by reducing its size, figuring out what is the field sitting at an offset from the start of a data structure, investigating ABI changes and more generally understanding a new codebase you have to work with.

It also uses these structure layouts to pretty print data feed to its standard input, e.g.:

```
$ pahole --header elf64_hdr --prettify /lib/modules/5.8.0-rc6+/build/vmlinux
{
        .e_ident = { 127, 69, 76, 70, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        .e_type = 2,
        .e_machine = 62,
        .e_version = 1,
        .e_entry = 16777216,
        .e_phoff = 64,
        .e_shoff = 604653784,
        .e_flags = 0,
        .e_ehsize = 64,
        .e_phentsize = 56,
        .e_phnum = 5,
        .e_shentsize = 64,
        .e_shnum = 80,
        .e_shstrndx = 79,
},
$
```

See the PRETTY PRINTING section for further examples and documentation.

The files must have associated debugging information. This information may be inside the file itself, in ELF sections, or in another file.

One way to have this information is to specify the **−g** option to the compiler when building it. When this is done the information will be stored in an ELF section. For the DWARF debugging information format this, adds, among others, the **.debug_info** ELF section. For CTF it is found in just one ELF section, **.SUNW_ctf**. BTF comes in at least the **.BTF** ELF section, and may come also with the **.BTF.ext** ELF section.

The **debuginfo** packages available in most Linux distributions are also supported by **pahole**, where the debugging information is available in a separate file.

By default, **pahole** shows the layout of all named structs in the files specified.

If no files are specified, then it will look if the /sys/kernel/btf/vmlinux is present, using the BTF information present in it about the running kernel, i.e. this works:

```
$ pahole list_head
struct list_head {
        struct list_head *      next;           /*     0     8 */
```

```
        struct list_head *      prev;           /*    8    8 */

        /* size: 16, cachelines: 1, members: 2 */
        /* last cacheline: 16 bytes */
};
$
```

If BTF is not present and no file is passed, then a vmlinux that matches the build-id for the running kernel will be looked up in the usual places, including where the kernel debuginfo packages put it, looking for DWARF info instead.

See the EXAMPLES section for more usage suggestions.

It also pretty prints whatever is fed to its standard input, according to the type specified, see the EXAMPLE session.

Use --count to state how many records should be pretty printed.

OPTIONS

pahole supports the following options.

−C, −−class_name=CLASS_NAMES
Show just these classes. This can be a comma separated list of class names or file URLs (e.g.: file://class_list.txt)

−c, −−cacheline_size=SIZE
Set cacheline size to SIZE bytes.

−−sort Sort the output by type name, maybe this will grow to allow sorting by other criteria.

This is mostly needed so that pretty printing from BTF and DWARF can be comparable when using using multiple threads to load DWARF data, when the order that the types in the compile units is processed is not deterministic.

−−count=COUNT
Pretty print the first COUNT records from input.

−−skip=COUNT
Skip COUNT input records.

−E, −−expand_types
Expand class members. Useful to find in what member of inner structs where an offset from the beginning of a struct is.

−F, −−format_path
Allows specifying a list of debugging formats to try, in order. Right now this includes "ctf" and "dwarf". The default format path used is equivalent to "-F dwarf,ctf".

−−hashbits=BITS
Allows specifying the number of bits for the debugging format loader to use. The only one affected so far is the "dwarf" one, its default now is 15, the maximum for it is now 21 bits. Tweak it

to see if it improves performance as the kernel evolves and more types and functions have to be loaded.

−−hex Print offsets and sizes in hexadecimal.

−r, −−rel_offset
Show relative offsets of members in inner structs.

−p, −−expand_pointers
Expand class pointer members.

−R, −−reorganize
Reorganize struct, demoting and combining bitfields, moving members to remove alignment holes and padding.

−S, −−show_reorg_steps
Show the struct layout at each reorganization step.

−i, −−contains=CLASS_NAME
Show classes that contains CLASS_NAME.

−a, −−anon_include
Include anonymous classes.

−A, −−nested_anon_include
Include nested (inside other structs) anonymous classes.

−B, −−bit_holes=NR_HOLES
Show only structs at least NR_HOLES bit holes.

−d, −−recursive
Recursive mode, affects several other flags.

−D, −−decl_exclude=PREFIX
exclude classes declared in files with PREFIX.

−f, −−find_pointers_to=CLASS_NAME
Find pointers to CLASS_NAME.

−H, −−holes=NR_HOLES
Show only structs with at least NR_HOLES holes.

−I, −−show_decl_info
Show the file and line number where the tags were defined, if available in the debugging information.

−−skip_encoding_btf_vars
Do not encode VARs in BTF.

−j, −−jobs=N
Run N jobs in parallel. Defaults to number of online processors + 10% (like the 'ninja' build system) if no argument is specified.

−J, −−btf_encode
Encode BTF information from DWARF, used in the Linux kernel build process when CONFIG_DEBUG_INFO_BTF=y is present, introduced in Linux v5.2. Used to implement features such as BPF CO-RE (Compile Once - Run Everywhere).

See *https://nakryiko.com/posts/bpf-portability-and-co-re/*.

−−btf_encode_detached=FILENAME
Same thing as -J/--btf_encode, but storing the raw BTF info into a separate file.

−−btf_encode_force
Ignore those symbols found invalid when encoding BTF.

−−btf_base=PATH
Path to the base BTF file, for instance: vmlinux when encoding kernel module BTF information. This may be inferred when asking for a /sys/kernel/btf/MODULE, when it will be autoconfigured to "/sys/kernel/btf/vmlinux".

−−btf_gen_floats
Allow producing BTF_KIND_FLOAT entries in systems where the vmlinux DWARF information has float types.

−−btf_gen_all
Allow using all the BTF features supported by pahole.

−l, −−show_first_biggest_size_base_type_member
Show first biggest size base_type member.

−m, −−nr_methods
Show number of methods of all classes, i.e. the number of functions have arguments that are pointers to a given class.

To get the number of methods for an specific class, please use:

```
$ pahole --nr_methods | grep -w sock
sock  1005
$
```

In the above example it used the BTF information in /sys/kernel/btf/vmlinux.

−M, −−show_only_data_members
Show only the members that use space in the class layout. C++ methods will be suppressed.

−n, −−nr_members
Show number of members.

−N, −−class_name_len
Show size of classes.

−O, −−dwarf_offset=OFFSET
Show tag with DWARF OFFSET.

−P, −−packable
Show only structs that has holes that can be packed if members are reorganized, for instance when using the **−−reorganize** option.

−P, −−with_flexible_array
Show only structs that have a flexible array.

−q, −−quiet
Be quieter.

−s, −−sizes
Show size of classes.

−t, −−separator=SEP
Use SEP as the field separator.

−T, −−nr_definitions
Show how many times struct was defined.

−u, −−defined_in
Show CUs where CLASS_NAME (-C) is defined.

−−flat_arrays
Flatten arrays, so that array[10][2] becomes array[20]. Useful when generating from both CTF/BTF and DWARF encodings for the same binary for testing purposes.

−−suppress_aligned_attribute
Suppress forced alignment markers, so that one can compare BTF or CTF output, that don’t have that info, to output from DWARF >= 5.

−−suppress_force_paddings

Suppress bitfield forced padding at the end of structs, as this requires something like DWARF’s DW_AT_alignment, so that one can compare BTF or CTF output, that don’t have that info.

−−suppress_packed

Suppress the output of the inference of __attribute__((__packed__)), so that one can compare BTF or CTF output, the inference algorithm uses things like DW_AT_alignment, so until it is improved to infer that as well for BTF, allow disabling this output.

−−fixup_silly_bitfields
Converts silly bitfields such as "int foo:32" to plain "int foo".

−V, −−verbose
be verbose

−−ptr_table_stats
Print statistics about ptr_table data structures, used to hold all the types, tags and functions data structures, for development tuning of such tables, tuned for a typical 2021 vmlinux file.

−w, −−word_size=WORD_SIZE
Change the arch word size to WORD_SIZE.

−x, −−exclude=PREFIX
Exclude PREFIXed classes.

−X, −−cu_exclude=PREFIX
Exclude PREFIXed compilation units.

−y, −−prefix_filter=PREFIX
Include PREFIXed classes.

−z, −−hole_size_ge=HOLE_SIZE
Show only structs with at least one hole greater or equal to HOLE_SIZE.

−−structs
Show only structs, all the other filters apply, i.e. to show just the sizes of all structs combine --structs with --sizes, etc.

−−packed
Show only packed structs, all the other filters apply, i.e. to show just the sizes of all packed structs combine --packed with --sizes, etc.

−−unions
Show only unions, all the other filters apply, i.e. to show just the sizes of all unions combine --union with --sizes, etc.

−−version
Show a traditional string version, i.e.: "v1.18".

−−numeric_version
Show a numeric only version, suitable for use in Makefiles and scripts where one wants to know what if the installed version has some feature, i.e.: 118 instead of "v1.18".

−−kabi_prefix=STRING
When the prefix of the string is STRING, treat the string as STRING.

NOTES

To enable the generation of debugging information in the Linux kernel build process select CONFIG_DEBUG_INFO. This can be done using make menuconfig by this path: "Kernel Hacking" -> "Compile-time checks and compiler options" -> "Compile the kernel with debug info". Consider as well enabling CONFIG_DEBUG_INFO_BTF by going thru the aforementioned menuconfig path and then selecting "Generate BTF typeinfo". Most modern distributions with eBPF support should come with that in all its kernels, greatly facilitating the use of pahole.

Many distributions also come with debuginfo packages, so just enable it in your package manager repository configuration and install the kernel-debuginfo, or any other userspace program written in a language that the compiler generates debuginfo (C, C++, for instance).

EXAMPLES

All the examples here use either /sys/kernel/btf/vmlinux, if present, or lookup a vmlinux file matching the running kernel, using the build-id info found in /sys/kernel/notes to make sure it matches.

Show a type:

```
$ pahole -C __u64
typedef long long unsigned int __u64;
$
```

Works as well if the only argument is a type name:

```
$ pahole raw_spinlock_t
typedef struct raw_spinlock raw_spinlock_t;
$
```

Multiple types can be passed, separated by commas:

```
$ pahole raw_spinlock_t,raw_spinlock
struct raw_spinlock {
        arch_spinlock_t         raw_lock;       /*     0     4 */

        /* size: 4, cachelines: 1, members: 1 */
        /* last cacheline: 4 bytes */
};
typedef struct raw_spinlock raw_spinlock_t;
$
```

Types can be expanded:

```
$ pahole -E raw_spinlock
struct raw_spinlock {
    /* typedef arch_spinlock_t */ struct qspinlock {
        union {
            /* typedef atomic_t */ struct {
                int counter;                                  /*     0     4 */
            } val;                                            /*     0     4 */
            struct {
                /* typedef u8 -> __u8 */ unsigned char locked;          /*     0     1 */
                /* typedef u8 -> __u8 */ unsigned char pending;         /*     1     1 */
            };                                                /*     0     2 */
            struct {
                /* typedef u16 -> __u16 */ short unsigned int locked_pending; /*     0     2 */
                /* typedef u16 -> __u16 */ short unsigned int tail;          /*     2     2 */
```

```
                };                                              /*     0     4 */
            };                                                  /*     0     4 */
        } raw_lock;                                             /*     0     4 */

        /* size: 4, cachelines: 1, members: 1 */
        /* last cacheline: 4 bytes */
};
$
```

When decoding OOPSes you may want to see the offsets and sizes in hexadecimal:

```
$ pahole --hex thread_struct
struct thread_struct {
        struct desc_struct      tls_array[3];   /*     0  0x18 */
        long unsigned int       sp;             /*  0x18   0x8 */
        short unsigned int      es;             /*  0x20   0x2 */
        short unsigned int      ds;             /*  0x22   0x2 */
        short unsigned int      fsindex;        /*  0x24   0x2 */
        short unsigned int      gsindex;        /*  0x26   0x2 */
        long unsigned int       fsbase;         /*  0x28   0x8 */
        long unsigned int       gsbase;         /*  0x30   0x8 */
        struct perf_event *     ptrace_bps[4];  /*  0x38  0x20 */
        /* --- cacheline 1 boundary (64 bytes) was 24 bytes ago --- */
        long unsigned int       debugreg6;      /*  0x58   0x8 */
        long unsigned int       ptrace_dr7;     /*  0x60   0x8 */
        long unsigned int       cr2;            /*  0x68   0x8 */
        long unsigned int       trap_nr;        /*  0x70   0x8 */
        long unsigned int       error_code;     /*  0x78   0x8 */
        /* --- cacheline 2 boundary (128 bytes) --- */
        struct io_bitmap *      io_bitmap;      /*  0x80   0x8 */
        long unsigned int       iopl_emul;      /*  0x88   0x8 */
        mm_segment_t            addr_limit;     /*  0x90   0x8 */
        unsigned int            sig_on_uaccess_err:1; /*  0x98: 0 0x4 */
        unsigned int            uaccess_err:1;  /*  0x98:0x1 0x4 */

        /* XXX 30 bits hole, try to pack */
        /* XXX 36 bytes hole, try to pack */

        /* --- cacheline 3 boundary (192 bytes) --- */
        struct fpu              fpu;            /*  0xc0 0x1040 */

        /* size: 4352, cachelines: 68, members: 20 */
        /* sum members: 4312, holes: 1, sum holes: 36 */
        /* sum bitfield members: 2 bits, bit holes: 1, sum bit holes: 30 bits */
};
$
```

OK, I know the offset that causes its a 'struct thread_struct' and that the offset is 0x178, so must be in that 'fpu' struct... No problem, expand 'struct thread_struct' and combine with **grep**:

```
$ pahole --hex -E thread_struct | egrep '(0x178|struct fpu)' -B4 -A4
        /* XXX 30 bits hole, try to pack */
        /* XXX 36 bytes hole, try to pack */

        /* --- cacheline 3 boundary (192 bytes) --- */
```

```
        struct fpu {
                unsigned int       last_cpu;                     /*  0xc0   0x4 */

                /* XXX 4 bytes hole, try to pack */

--
                        /* typedef u8 -> __u8 */ unsigned char alimit;          /* 0x171   0x1 */

                        /* XXX 6 bytes hole, try to pack */

                        struct math_emu_info * info;                         /* 0x178   0x8 */
                        /* --- cacheline 6 boundary (384 bytes) --- */
                        /* typedef u32 -> __u32 */ unsigned int entry_eip;       /* 0x180   0x4 */
                } soft; /* 0x100  0x88 */
                struct xregs_state {
$
```

Want to know where 'struct thread_struct' is defined in the kernel sources?

```
$ pahole -I thread_struct | head -2
/* Used at: /sys/kernel/btf/vmlinux */
/* <0> (null):0 */
$
```

Not present in BTF, so use DWARF, takes a little bit longer, and assuming it finds the matching vmlinux file:

```
$ pahole -Fdwarf -I thread_struct | head -2
/* Used at: /home/acme/git/linux/arch/x86/kernel/head64.c */
/* <3333> /home/acme/git/linux/arch/x86/include/asm/processor.h:485 */
$
```

To find the biggest data structures in the Linux kernel:

```
$ pahole -s | sort -k2 -nr | head -5
cmp_data              290904 1
dec_datas             274520 1
cpu_entry_area         217088 0
pglist_data           172928 4
saved_cmdlines_buffer  131104 1
$
```

The second column is the size in bytes and the third is the number of alignment holes in that structure.

Show data structures that have a raw spinlock and are related to the RCU mechanism:

```
$ pahole --contains raw_spinlock_t --prefix rcu
rcu_node
rcu_data
rcu_state
$
```

To see that in context, combine it with *grep*:

```
$ pahole rcu_state | grep raw_spinlock_t -B1 -A5
        /* --- cacheline 52 boundary (3328 bytes) --- */
        raw_spinlock_t          ofl_lock;           /*  3328     4 */
```

```
        /* size: 3392, cachelines: 53, members: 35 */
        /* sum members: 3250, holes: 7, sum holes: 82 */
        /* padding: 60 */
};
$
```

PRETTY PRINTING

pahole can also use the data structure types to pretty print raw data specified via --prettify. To consume raw data from the standard input, just use '--prettify -'

It can also pretty print raw data from stdin according to the type specified:

```
$ pahole -C modversion_info drivers/scsi/sg.ko
struct modversion_info {
    long unsigned int        crc;              /*     0     8 */
    char                     name[56];         /*     8    56 */

    /* size: 64, cachelines: 1, members: 2 */
};
$
$ objcopy -O binary --only-section=__versions drivers/scsi/sg.ko versions
$
$ ls -la versions
-rw-rw-r--. 1 acme acme 7616 Jun 25 11:33 versions
$
$ pahole --count 3 -C modversion_info drivers/scsi/sg.ko --prettify versions
{
    .crc = 0x8dabd84,
    .name = "module_layout",
},
{
    .crc = 0x45e4617b,
    .name = "no_llseek",
},
{
    .crc = 0xa23fae8c,
    .name = "param_ops_int",
},
$
$ pahole --skip 1 --count 2 -C modversion_info drivers/scsi/sg.ko --prettify - < versions
{
    .crc = 0x45e4617b,
    .name = "no_llseek",
},
{
    .crc = 0xa23fae8c,
    .name = "param_ops_int",
},
$
```

This is equivalent to:

```
$ pahole --seek_bytes 64 --count 1 -C modversion_info drivers/scsi/sg.ko --prettify versions
{
        .crc = 0x45e4617b,
        .name = "no_llseek",
```

```
},
$
```

−C, −−class_name=CLASS_NAME

Pretty print according to this class. Arguments may be passed to it to affect how the pretty printing is performed, e.g.:

```
-C 'perf_event_header(sizeof,type,type_enum=perf_event_type,filter=type==PERF_RECORD_EXIT)'
```

This would select the 'struct perf_event_header' as the type to use to pretty print records states that the 'size' field in that struct should be used to figure out the size of the record (variable sized records), that the 'enum perf_event_type' should be used to pretty print the numeric value in perf_event_header->type and furthermore that it should be used to heuristically look for structs with the same name (lowercase) of the enum entry that is converted from the type field, using it to pretty print instead of the base 'perf_event_header' type. See the PRETTY PRINTING EXAMPLES section below.

Furthermore the 'filter=' part can be used, so far with only the '==' operator to filter based on the 'type' field and converting the string 'PERF_RECORD_EXIT' to a number according to type_enum.

The 'sizeof' arg defaults to the 'size' member name, if the name is different, one can use 'sizeof=sz' form, ditto for 'type=other_member_name' field, that defaults to 'type'.

PRETTY PRINTING EXAMPLES

Looking at the ELF header for a vmlinux file, using BTF, first lets discover the ELF header type:

```
$ pahole --sizes | grep -i elf | grep -i _h
elf64_hdr          64        0
elf32_hdr          52        0
$
```

Now we can use this to show the first record from offset zero:

```
$ pahole -C elf64_hdr --count 1 --prettify /lib/modules/5.8.0-rc3+/build/vmlinux
{
        .e_ident = { 127, 69, 76, 70, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        .e_type = 2,
        .e_machine = 62,
        .e_version = 1,
        .e_entry = 16777216,
        .e_phoff = 64,
        .e_shoff = 775923840,
        .e_flags = 0,
        .e_ehsize = 64,
        .e_phentsize = 56,
        .e_phnum = 5,
        .e_shentsize = 64,
        .e_shnum = 80,
        .e_shstrndx = 79,
},
$
```

This is equivalent to:

```
$ pahole --header elf64_hdr --prettify /lib/modules/5.8.0-rc3+/build/vmlinux
```

The --header option also allows reference in other command line options to fields in the header. This is useful when one wants to show multiple records in a file and the range where those fields are located is specified in header fields, such as for perf.data files:

```
$ pahole --hex ~/bin/perf --header perf_file_header --prettify perf.data
```

```
{
        .magic = 0x32454c4946524550,
        .size = 0x68,
        .attr_size = 0x88,
        .attrs = {
                .offset = 0xa8,
                .size = 0x88,
        },
        .data = {
                .offset = 0x130,
                .size = 0x588,
        },
        .event_types = {
                .offset = 0,
                .size = 0,
        },
        .adds_features = { 0x16717ffc, 0, 0, 0 },
},
$
```

So to display the cgroups records in the perf_file_header.data section we can use:

```
$ pahole ~/bin/perf --header=perf_file_header --seek_bytes '$header.data.offset' --size_bytes='$header.data.size' -C 'perf_
{
        .header = {
                .type = PERF_RECORD_CGROUP,
                .misc = 0,
                .size = 40,
        },
        .id = 1,
        .path = "/",
},
{
        .header = {
                .type = PERF_RECORD_CGROUP,
                .misc = 0,
                .size = 48,
        },
        .id = 1553,
        .path = "/system.slice",
},
{
        .header = {
                .type = PERF_RECORD_CGROUP,
                .misc = 0,
                .size = 48,
        },
        .id = 8,
        .path = "/machine.slice",
},
{
        .header = {
                .type = PERF_RECORD_CGROUP,
                .misc = 0,
                .size = 128,
        },
```

```
        .id = 7828,
        .path = "/machine.slice/libpod-42be8e8d4eb9d22405845005f0d04ea398548dccc934a150fbaa3c1f1f9492c2.scop
},
{
        .header = {
                .type = PERF_RECORD_CGROUP,
                .misc = 0,
                .size = 88,
        },
        .id = 13,
        .path = "/machine.slice/machine-qemu\x2d1\x2drhel6.sandy.scope",
},
$
```

For the common case of the header having a member that has the ’offset’ and ’size’ members, it is possible to use this more compact form:

```
$ pahole ~/bin/perf --header=perf_file_header --range=data -C ’perf_event_header(sizeof,type,type_enum=perf_event_typ
```

This uses ~/bin/perf to get the type definitions, the defines ’struct perf_file_header’ as the header, then seeks ’$header.data.offset’ bytes from the start of the file, and considers ’$header.data.size’ bytes worth of such records. The filter expression may omit a common prefix, in this case it could additionally be equivalently written as both ’filter=type==CGROUP’ or the ’filter=’ can also be omitted, getting as compact as ’type==CGROUP’:

If we look at:

```
$ pahole ~/bin/perf -C perf_event_header
struct perf_event_header {
        __u32                   type;                 /*     0     4 */
        __u16                   misc;                 /*     4     2 */
        __u16                   size;                 /*     6     2 */

        /* size: 8, cachelines: 1, members: 3 */
        /* last cacheline: 8 bytes */
};
$
```

And:

```
$ pahole ~/bin/perf -C perf_event_type
enum perf_event_type {
        PERF_RECORD_MMAP = 1,
        PERF_RECORD_LOST = 2,
        PERF_RECORD_COMM = 3,
        PERF_RECORD_EXIT = 4,
        PERF_RECORD_THROTTLE = 5,
        PERF_RECORD_UNTHROTTLE = 6,
        PERF_RECORD_FORK = 7,
        PERF_RECORD_READ = 8,
        PERF_RECORD_SAMPLE = 9,
        PERF_RECORD_MMAP2 = 10,
        PERF_RECORD_AUX = 11,
        PERF_RECORD_ITRACE_START = 12,
        PERF_RECORD_LOST_SAMPLES = 13,
        PERF_RECORD_SWITCH = 14,
        PERF_RECORD_SWITCH_CPU_WIDE = 15,
        PERF_RECORD_NAMESPACES = 16,
        PERF_RECORD_KSYMBOL = 17,
```

```
        PERF_RECORD_BPF_EVENT = 18,
        PERF_RECORD_CGROUP = 19,
        PERF_RECORD_TEXT_POKE = 20,
        PERF_RECORD_MAX = 21,
};
$
```

And furthermore:

```
$ pahole ~/bin/perf -C perf_record_cgroup
struct perf_record_cgroup {
        struct perf_event_header   header;              /*     0     8 */
        __u64                      id;                  /*     8     8 */
        char                       path[4096];          /*    16  4096 */

        /* size: 4112, cachelines: 65, members: 3 */
        /* last cacheline: 16 bytes */
};
$
```

Then we can see how the perf_event_header.type could be converted from a __u32 to a string (PERF_RECORD_CGROUP). If we remove that type_enum=perf_event_type, we will lose the conversion of 'struct perf_event_header' to the more descriptive 'struct perf_record_cgroup', and also the beautification of the header.type field:

```
$ pahole ~/bin/perf --header=perf_file_header --seek_bytes '$header.data.offset' --size_bytes='$header.data.size' -C 'perf_
{
        .type = 19,
        .misc = 0,
        .size = 40,
},
{
        .type = 19,
        .misc = 0,
        .size = 48,
},
{
        .type = 19,
        .misc = 0,
        .size = 48,
},
{
        .type = 19,
        .misc = 0,
        .size = 128,
},
{
        .type = 19,
        .misc = 0,
        .size = 88,
},
$
```

Some of the records are not found in 'type_enum=perf_event_type' so some of the records don't get converted to a type that fully shows its contents. For perf we know that those are in another enumeration, 'enum perf_user_event_type', so, for these cases, we can create a 'virtual enum', i.e. the sum of two enums and then get all those entries decoded and properly casted, first few records with just 'enum

perf_event_type’:

$ pahole ~/bin/perf --header=perf_file_header --seek_bytes ’$header.data.offset’ --size_bytes=’$header.data.size’ -C ’perf_
```
{
	.type = 79,
	.misc = 0,
	.size = 32,
},
{
	.type = 73,
	.misc = 0,
	.size = 40,
},
{
	.type = 74,
	.misc = 0,
	.size = 32,
},
{
	.header = {
		.type = PERF_RECORD_CGROUP,
		.misc = 0,
		.size = 40,
	},
	.id = 1,
	.path = "/",
},
$
```

Now with both enumerations, i.e. with ’type_enum=perf_event_type+perf_user_event_type’:

$ pahole ~/bin/perf --header=perf_file_header --seek_bytes ’$header.data.offset’ --size_bytes=’$header.data.size’ -C ’perf_
```
{
	.header = {
		.type = PERF_RECORD_TIME_CONV,
		.misc = 0,
		.size = 32,
	},
	.time_shift = 31,
	.time_mult = 1016803377,
	.time_zero = 435759009518382,
},
{
	.header = {
		.type = PERF_RECORD_THREAD_MAP,
		.misc = 0,
		.size = 40,
	},
	.nr = 1,
	.entries = 0x50 0x7e 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x
},
{
	.header = {
		.type = PERF_RECORD_CPU_MAP,
		.misc = 0,
		.size = 32,
```

```
		},
		.data = {
			.type = 1,
			.data = "",
		},
},
{
		.header = {
			.type = PERF_RECORD_CGROUP,
			.misc = 0,
			.size = 40,
		},
		.id = 1,
		.path = "/",
},
{
		.header = {
			.type = PERF_RECORD_CGROUP,
			.misc = 0,
			.size = 48,
		},
		.id = 1553,
		.path = "/system.slice",
},
$
```

It is possible to pass multiple types, one has only to make sure they appear in the file in sequence, i.e. for the perf.data example, see the perf_file_header dump above, one can print the perf_file_attr structs in the header attrs range, then the perf_event_header in the data range with the following command:

```
pahole ~/bin/perf --header=perf_file_header -C 'perf_file_attr(range=attrs),perf_event_header(range=data,sizeof,type
```

SEE ALSO

eu-readelf(1), *readelf*(1), *objdump*(1).

https://www.kernel.org/doc/ols/2007/ols2007v2-pages-35-44.pdf.

AUTHOR

pahole was written and is maintained by Arnaldo Carvalho de Melo <acme@kernel.org>.

Thanks to Andrii Nakryiko and Martin KaFai Lau for providing the BTF encoder and improving the codebase while making sure the BTF encoder works as needed to be used in encoding the Linux kernel .BTF section from the DWARF info generated by gcc. For that Andrii wrote a BTF deduplicator in libbpf that is used by **pahole**.

Also thanks to Conectiva, Mandriva and Red Hat for allowing me to work on these tools.

Please send bug reports to <dwarves@vger.kernel.org>.

No subscription is required.